

MSC. THESIS ECONOMETRICS AND MANAGEMENT SCIENCE

SPECIALISATION: BUSINESS ANALYTICS AND QUANTITATIVE
MARKETING

Estimating numbers of refugees in refugee camps using satellite imagery

Name student:

P.J.P. (PAUL) KOOT

Student ID number:

483027

Supervisor EUR:

PROF.DR. S.I. (ILKER) BIRBIL

Supervisor Notilyze:

F.P.N. (COLIN) NUGTEREN

Co-reader EUR:

DR. F. (FLAVIUS) FRASINCAR



The content of this thesis is the sole responsibility of the author and does not reflect the view of either
Erasmus School of Economics or Erasmus University.

April 3, 2020

Contents

Acknowledgements	iii
List of Acronyms	iv
List of Figures	v
List of Tables	x
1 Introduction	1
2 Literature review	4
3 Data	8
3.1 Data Description	8
3.2 Data Descriptives	10
3.3 Data Quality	13
4 Methodology	15
4.1 Preprocessing	15
4.2 Neural Networks	16
4.3 Faster R-CNN model	28
4.4 Boosting	43
5 Results	50
5.1 Results Benchmark Model	50
5.2 Boosting Results	62
5.3 Linear Regression for Population Estimation	66
6 Discussion	74
7 Recommendations and Possible Extensions	77
References	82
Appendices	85
A Surveys Data Fields	85
B Pre-training and Tent Detection	88
C Mean-Shift Clustering	94

4 Methodology

This section starts with the preprocessing of the model input (Section 4.1). This is an important part of the contribution to the literature. We explore ways in which satellite imagery can be preprocessed in order to obtain a better performing model. To our knowledge, there have been little discussion or documentation on how to preprocess these kind of images. Especially the fact that different satellite images can come with different zoom levels, which are often saved as well, this kind of information can be used to improve the model. As already discussed in Section 2 CNNs are the state of the art models in image classification since the breakthrough of AlexNet in 2012 (Krizhevsky, Sutskever, and G. E. Hinton, 2012). In the meantime, research has also shown that besides classification CNNs are useful in general for imagery tasks. See e.g. Long, Shelhamer, and Darrell (2015), who show CNNs also outperform the state-of-the-art semantic segmentation. So it is a logical choice to use a CNN to extract information from the satellite images. In the rest of the section the used model and hyperparameter settings are discussed. In Section 4.2 Neural Networks are being discussed, which is followed by the introduction of the Faster R-CNN model (Section 4.3). This specific type of Neural Networks is the state of the art in image processing, as can be read in Section 2. We conclude this chapter with our implementation of a Boosting method in Section 4.4.

4.1 Preprocessing

As a starting point of the data preparation we take the survey rounds in the camps (see Table 1). The 74 surveys that are exactly identical in number of IDPs are removed from the initial dataset, as the population in these kind of camps is very dynamic and these data points are most likely prone to errors. Also during the whole preprocessing we take into account that the model must be robust to human input. For example we deal with different zoom levels as input, different resolutions, no clear refugee camp borders and Google Earth logos in an image.

After further inspection, of the 110 refugee camps in which surveys have been done, 56 camps are selected for further analysis. This selection is based on both information from the surveys and satellite imagery. The most important reason to exclude camps is the fact that they cannot be identified visually on satellite images. A reason for this can be found in the survey data, where certain camps are classified as ‘host community/families’. These ‘camps’ are spread over current houses and families, which makes it impossible to detect from satellites without knowing what houses are part of this community. Another reason is that for some camps there are certain groups of buildings/tents of which it is clear that they are part of the camp, but the borders of the camp are unclear. If it is already unclear to determine what houses are part of the camp by eye, it is difficult to train a computer vision model that can. After these filters we end up with 638

survey data points across 56 camps.

The next step is to determine the boundaries of the camps, such that the whole camp is included in one image. This is done by visual inspection, using the geographic location of the camp that is available in the survey data. Making the whole camp included within one satellite image comes along with a specific zoom level for each camp. Visual inspection and consequently manual extraction of the images is done to see if the model can be made generalisable for multiple zoom levels. An advantage of this is that the model becomes more robust and is less dependent on the zoom level of new images of new camps. This makes the model more user-friendly. Also it is useful to see what range of zoom levels are detectable by the model and to see for what object size the model works best. Based on the available survey information of each camp, several satellite images of one camp are included in the dataset from Google Earth: these images have a good resolution of around 1 m/pixel. However we should remark that the frequency of these images is low and very dependent on the location.

An image is matched to a survey of a camp based on the following rules:

- Determine the approximate survey date, which is defined as the middle of the interval the survey took place in (see Table 1 for these intervals).
- Find the image closest to this approximate survey date, with the restriction that the image should always originate from a date earlier than this approximate survey date.

Especially this final step might introduce some bias in the estimation of the number of IDPs in a camp. However, an effort is made to take this time shift into account by keeping track of this difference in timing of survey and satellite image and use this difference as a variable in the final regression.

4.2 Neural Networks

This section is structured as follows. First of all, we give a conceptual explanation of what a neural network is and where the idea of a neural network comes from. Then, we will examine the maths behind a neural network, making sure that such a network can learn from observations. This process of learning is called backpropagation. Then, we will move on to CNNs. We explain what specific layers characterises a CNN and we also explain how these layers process images. Then we will also touch upon the process of backpropagation in such a CNN. Especially the problem of vanishing gradients and how this is less of a problem in backpropagation in CNNs is examined in this final part.

A neural network can be seen as a machine learning algorithm that is based on how our brains work. We learn to distinguish between cats and dogs by seeing some of both while our parents, teacher or friends tell us what we see is either a cat or a dog. We ‘update our brains’ with this information and memorise it. After a while we can distinguish between cats and dogs ourselves, based on several features that distinguish both animals; e.g. the whiskers of a cat or the large ears of a dog. The same is done by a neural network; we can learn a neural network to forecast an observation based on features it gets as an input. Instead of brains to save information, the neural network has nodes and weights. The nodes are divided into layers and these layers can be connected in different ways with weights. In a fully connected layer all nodes of one layer are connected with the nodes of a next layer, making sure that the output of any node in layer 1 can activate any node in layer 2. Activation of nodes often happens with activation functions. Nowadays the most used one is the Rectified Linear Unit (ReLU) layer, that is linearly activated after a certain threshold has been reached by the sum of weighted inputs from the previous layer. The weights between the connections of the nodes are trained by supervised training. A set of input features and corresponding known output are fed into the neural network and the network adjust its weights to minimise a specified loss function. This loss function represents the difference in predicted output and known output and is minimised by adapting the weights via gradient descent. The difference in loss when changing weight values is determined by error backpropagation. Weights that decrease the error term most when changed, are changed most and weights that do not influence the error when they are changed, are not influenced or less. In this way we can update the weights for every (batch of) known observations.

Backpropagation

An important part of the neural network design is the way it is being trained to minimise the loss function. As loss function we use the cross entropy function to deal with the classification of objects. Our loss function for classification is defined as:

$$E(\beta) = - \sum_{i=1}^n \sum_{k=1}^K (y_{ik} \ln(\hat{y}_{ik}) + (1 - y_{ik}) \ln(1 - \hat{y}_{ik})), \quad (1)$$

where $\hat{y}_{ik} = \hat{y}_k(x_i, \beta)$. K is the number of classes of the problem, y_i is the real observation, which is coded as a dummy vector of length K filled with zeros and one 1 that indicates the class corresponding to the observation. This means the K classes have to be ordered in an arbitrary, but consistent order. Then we predict the class of an observation i with the prediction \hat{y}_i , which is again a vector of length K with for each position k an estimated probability that observation i belongs to class k . These probabilities sum up to 1 and one element of the vector can be extracted with the notation \hat{y}_{ik} . x_i is the vector of features of observation i , that we want to use to predict \hat{y}_i . Then β indicates the vector of regressors that is used to describe the relation between the features x_i and the prediction \hat{y}_i . In the context of a neural network, x_i

is the input at the input layer, β represents the weights between nodes and \hat{y}_i is the final outcome of the model. Next, we will explain the function $\hat{y}_k(x_i, \beta)$.

In Figure 7 a sample neural network is shown. In this figure the following equations are applicable:

$$\hat{y}_k = \sigma(a_k), \quad a_k = \sum_{m=0}^M \beta_{km}^{(2)} z_m, \quad z_m = h(a_m), \quad a_m = \sum_{p=0}^P \beta_{mp}^{(1)} x_p.$$

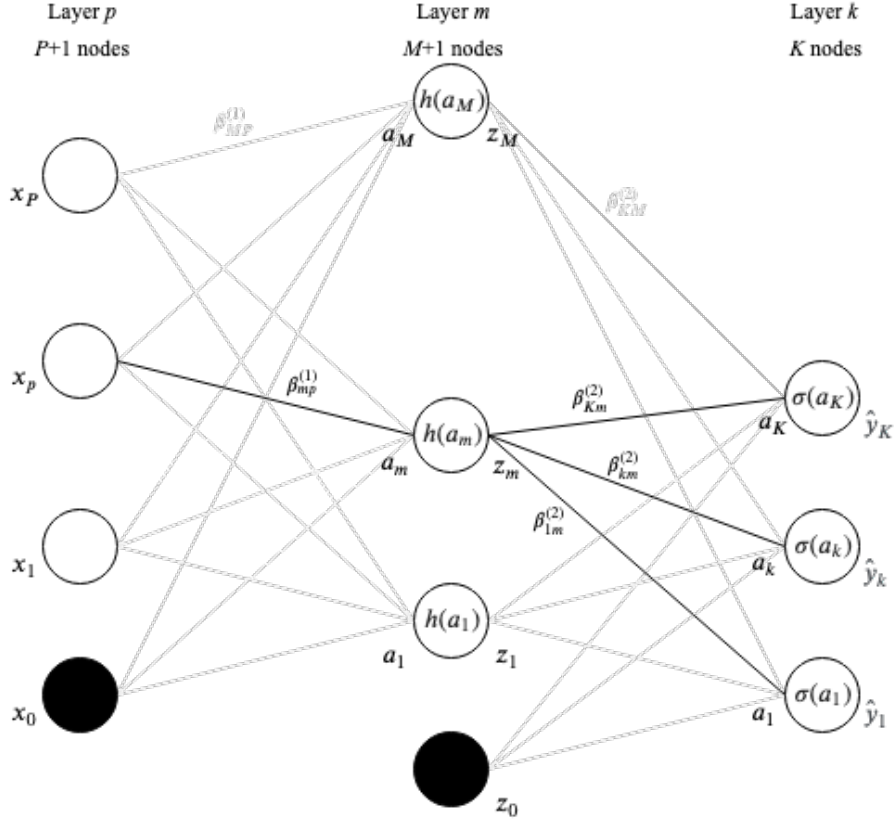


Figure 7: A neural network with one hidden layer. This can be generalised to an arbitrary number of hidden layers, but for derivation purposes only one hidden layer has been implemented in this example.

To determine what weights need to be changed in order for the error function to be minimised, we have to determine the derivative of the error with respect to each weight. Following our example in Figure 7, this can be done as follows. We start with the error and backpropagate our differentiation up to the weight which we want to investigate. For the final layer we look at $\beta_{km}^{(2)}$ (middle black weight connection in the second spacing in Figure 7). As we know from Equation 1, $E_i(\beta)$ is dependent on \hat{y}_{ik} and we know for one observation i .

We can express \hat{y}_{ik} in terms of $\beta_{km}^{(2)}$ as follows:

$$\hat{y}_k = \sigma(a_k) = \sigma\left(\sum_{m=0}^M \beta_{km}^{(2)} z_m\right). \quad (2)$$

We can rewrite the derivative to

$$\frac{\partial E_i(\beta)}{\partial \beta_{km}^{(2)}} = \frac{\partial E_i}{\partial a_k} \frac{\partial a_k}{\partial \beta_{km}^{(2)}}. \quad (3)$$

As can be seen in the chain of partial derivatives, the backpropagation starts with the derivative of the error function on the final activation value of a class a_k (before activation). We can rewrite this term by combining this term $\left(\frac{\partial E_i(\beta)}{\partial a_k}\right)$ with our loss function (Equation 1):

$$\begin{aligned} \frac{\partial E_i(\beta)}{\partial a_k} &= \frac{\partial E_{ik}}{\partial a_k} \\ &= \frac{\partial}{\partial a_k} (-(y_{ik} \ln(\hat{y}_{ik}) + (1 - y_{ik}) \ln(1 - \hat{y}_{ik}))) \\ &= -y_{ik} \frac{\partial}{\partial a_k} (\ln(\hat{y}_{ik})) - (1 - y_{ik}) \frac{\partial}{\partial a_k} (\ln(1 - \hat{y}_{ik})). \end{aligned} \quad (4)$$

As stated before, $\hat{y}_{ik} = \hat{y}_k(x_i, \beta)$. For the final layer we use a softmax layer, which boils down to a logit layer as we only deal with the classes ‘tent’ and ‘background’. This gives:

$$\hat{y}_k(x_i, \beta) = \frac{e^{a_k}}{1 + e^{a_k}} = \frac{1}{1 + e^{-a_k}}, \quad (5)$$

$$1 - \hat{y}_k(x_i, \beta) = 1 - \frac{1}{1 + e^{-a_k}} = \frac{e^{-a_k}}{1 + e^{-a_k}}, \quad (6)$$

where $a_k = \sum_{m=0}^M \beta_{km}^{(L-1)} z_m$ and $L - 1 = 2$.

Note that z_m is the outcome of node m of the previous layer which is ultimately dependent on the input features x_i , hence the notation $\hat{y}_k(x_i, \beta)$. The superscript $(L-1)$ indicates that this weight β_{km} is defined in the $(L-1)^{th}$ space between two layers, i.e. in the final space, just before layer L . Continuing with Equation 4 after substitution of (5) and (6) gives:

$$\begin{aligned}
\frac{\partial E_i(\beta)}{\partial a_k} &= -y_{ik} \frac{\partial}{\partial a_k} (\ln(\hat{y}_{ik})) - (1 - y_{ik}) \frac{\partial}{\partial a_k} (\ln(1 - \hat{y}_{ik})) \\
&= -y_{ik} \frac{\partial}{\partial a_k} \left(\ln \left(\frac{1}{1 + e^{-a_k}} \right) \right) + (y_{ik} - 1) \left(\frac{\partial}{\partial a_k} \left(\ln \left(\frac{e^{-a_k}}{1 + e^{-a_k}} \right) \right) \right) \\
&= -y_{ik} \frac{\partial}{\partial a_k} (\ln(1) - \ln(1 + e^{-a_k})) + (y_{ik} - 1) \left(\frac{\partial}{\partial a_k} (\ln(e^{-a_k}) - \ln(1 + e^{-a_k})) \right) \\
&= -y_{ik} \frac{e^{-a_k}}{1 + e^{-a_k}} + (y_{ik} - 1) \left(-1 + \frac{e^{-a_k}}{1 + e^{-a_k}} \right) \\
&= -y_{ik} \frac{e^{-a_k}}{1 + e^{-a_k}} + y_{ik} \left(-1 + \frac{e^{-a_k}}{1 + e^{-a_k}} \right) - 1 \left(-1 + \frac{e^{-a_k}}{1 + e^{-a_k}} \right) \\
&= -y_{ik} - \left(-1 + \frac{e^{-a_k}}{1 + e^{-a_k}} \right) \\
&= 1 - \frac{e^{-a_k}}{1 + e^{-a_k}} - y_{ik} \\
&= \frac{1}{1 + e^{-a_k}} - y_{ik} \\
&= \hat{y}_k(x_i, \beta) - y_{ik} \\
&= \hat{y}_{ik} - y_{ik} \equiv \delta_k
\end{aligned}$$

Note that this case can be generalised for multinomial classification problems with K classes, as is the case in Figure 7. For this case we have a softmax activation function:

$$\hat{y}_k(x_i, \beta) = \frac{e^{a_k}}{\sum_{k=1}^K e^{a_k}} = \frac{e^{a_k}}{1 + \sum_{k=1}^{K-1} e^{-a_k}}, \quad (7)$$

where the 1 in the denominator occurs again by setting all weights $[\beta_{K,0} \dots \beta_{K,M}] = 0$ as the weights are not determined independently. Due to the restriction $\sum_{k=1}^K \hat{y}_{ik} = 1$ all weights are determined relatively to a base case, just as for the sigmoid function. When differentiating Equation 7 with respect to a_k , all classes from the summation cancel out except for class k , leading again to Equation 5 and 6.

The second term $\left(\frac{\partial a_k(\beta)}{\partial \beta_{km}^{(2)}} \right)$ is easier to derive, as $a_k(\beta) = \sum_{m=0}^M \beta_{km}^{(2)} z_m$, which gives $\frac{\partial a_k(\beta)}{\partial \beta_{km}^{(2)}} = z_m$. This gives

$$\frac{\partial E_i(\beta)}{\partial \beta_{km}^{(2)}} = \frac{\partial E_i}{\partial a_k} \frac{\partial a_k}{\partial \beta_{km}^{(2)}} = (\hat{y}_{ik} - y_i) z_m = \delta_k z_m.$$

To obtain the derivative of the loss function with respect to weights in the first layer, we inspect $\beta_{mp}^{(1)}$. To get this derivative, we have to rewrite Equation 2:

$$\hat{y}_k = \sigma(a_k) = \sigma \left(\sum_{m=0}^M \beta_{km}^{(2)} z_m \right) = \sigma \left(\sum_{m=0}^M \beta_{km}^{(2)} (h(a_m)) \right) = \sigma \left(\sum_{m=0}^M \beta_{km}^{(2)} \left(h \left(\sum_{p=0}^P \beta_{mp}^{(1)} x_p \right) \right) \right). \quad (8)$$

This gives

$$\frac{\partial E_i(\beta)}{\partial \beta_{mp}^{(1)}} = \frac{\partial E_i}{\partial a_m} \frac{\partial a_m}{\partial \beta_{mp}^{(1)}}. \quad (9)$$

Note that a_m will influence the error E_i via more than one path (see the three black paths in Figure 7).

This leads to

$$\frac{\partial E_i}{\partial a_m} = \sum_{k=1}^K \frac{\partial E_i}{\partial a_k} \frac{\partial a_k}{\partial a_m} = \sum_{k=1}^K \frac{\partial E_i}{\partial a_k} h'(a_m) \beta_{km} = h'(a_m) \sum_{k=1}^K (\hat{y}_{ik} - y_i) \beta_{km} = h'(a_m) \sum_{k=1}^K \delta_k \beta_{km} = \delta_m.$$

The second term $\left(\frac{\partial a_m}{\partial \beta_{mp}^{(1)}} \right)$ with $a_m(\beta) = \sum_{p=0}^P \beta_{mp}^{(2)} x_p$ gives $\frac{\partial a_m}{\partial \beta_{mp}^{(1)}} = x_p$. Continuing with Equation 9, we obtain

$$\frac{\partial E_i(\beta)}{\partial \beta_{mp}^{(1)}} = \frac{\partial E_i}{\partial a_m} \frac{\partial a_m}{\partial \beta_{mp}^{(1)}} = h'(a_m) \sum_{k=1}^K \delta_k \beta_{km} x_p = \delta_m x_p. \quad (10)$$

We could do this iteratively with more hidden layers. Note that δ_k is in the expression for δ_m : the error at node k in this way backpropagates towards the first layer. Also note we only need the input x_p , the intermediate results z_m and the real and predicted values, respectively y_{ik} and \hat{y}_{ik} , to calculate all gradients. Each training epoch, all these values are calculated with forward propagation: we give the input vector \mathbf{x} and we obtain the other mentioned values. Then, using backward propagation as derived above, we can calculate all necessary gradients. One more difficult part to calculate is the term $h'(a_m)$. However in the next section we will see how our model is dealing with this potential problem. Also note that during the derivation of the backpropagation we assumed we only trained the model on one observation by taking $E_i(\beta)$. The neural network can be trained on all samples N by using e.g. $\sum_{i=1}^N \frac{\partial E_i(\beta)}{\partial \beta_{km}^{(2)}}$ instead of $\frac{\partial E_i(\beta)}{\partial \beta_{km}^{(2)}}$ in Equation 3 (Batch Gradient Descent). Another variant is Mini-Batch Gradient Descent, in which training is done on a subsample of the training set every epoch. However, as our model will train on one picture at a time, we derived everything based on this (Stochastic Gradient Descent). With the calculated gradients, the parameter update for $\beta_{mp}^{(1)}$ at iteration $t + 1$ follows with

$$\beta_{mp}^{(1),t+1} = \beta_{mp}^{(1),t} - \gamma \frac{\partial E_i(\beta^t)}{\partial \beta_{mp}^{(1)}} = \beta_{mp}^{(1)} - \gamma \delta_m x_p, \quad (11)$$

where γ represents the learning rate. This hyperparameter can be tuned manually.

Our model is implemented with a slightly different version of SGD, called ‘Adam’ (Kingma and Ba, 2014). This algorithm uses the gradient to update the parameters, however it takes into account the current uncertainty of the gradient by considering the second moment (a measure of the uncentered variance) of the

gradient in the previous steps. A larger variance means a smaller step size. Compared to SGD Adam has the advantage of being able to adapt learning rates for each parameter individually. However, Adam also has disadvantages compared to SGD. The main reason against using Adam is that it generalises poorly compared to SGD (Keskar and Socher, 2017). This problem is currently subject of research, but an explanation could be that due to the fact Adam adapts its learning rate based on momentum, it will take steps more carefully, which could lead to cases in which Adam is stuck in a local minimum more quickly.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have some specific properties that are useful for image classification. An image can be represented as a matrix of input values, e.g. by representing every pixel of an image in RGB values. To recognise if an image is a cat or a dog, we could train a normal neural network. However, when doing so, a few problems arise. First of all, every node of the first layer of a neural network (the input layer) can take one value as input (if we want to use the information in each pixel individually). For a relative small image of 60×60 pixels with RGB values, we already need 10,800 nodes in the first layer. Classic neural networks used for image classification become large very quickly due to this large amount of input features. Adding some extra fully connected hidden layers will cause the amount of weights to be trained to grow quickly, resulting in the need of huge amount of training images to get a moderate performance. Also, when a cat or dog is just another colour, or another background, or another location in the picture, this all needs to be in the training set in order to be recognised by the neural network.

Instead, convolutional neural networks use convolution layers. These layers can be seen as a kind of filters that scan the image. A 3×3 convolution layer has 9 weights that are activated by certain patterns in a 3×3 part of the image. An advantage of this structure is that one (set of) layers that for example extract squares from images, can do that at any location of the image, because the layer is striding along the image. Also the layer is less sensitive to larger and smaller squares in images due to the downsampling and therefore compression of the information. This gives an intuitive reason for the fact that CNNs needs less parameters than fully connected neural networks, who are not that flexible when it comes to the location or size of an object. Using these convolution layers sequentially, the first layers will recognise a boundary in pixel values as a contour, while some layers later on in the network will combine these activated contour layers and learn that they are boxes, or triangles, etc. In Figure 8 the mechanism behind a convolution layer is shown. Convolution layers have three properties: the kernel size K (e.g. a 3×3 filter), the stride S which is the step size of the filter and the padding P . A padding value $P = 1$ means that the input image is surrounded with one row/column with zeros (which would make the input image in Figure 8 of size 14×14). This causes the output after convolution to shrink less or not to shrink at all, depending on the chosen stride and kernel

size. The three parameters are summarised by the notation (K, S, P) . For example, Figure 8 shows a $(3,1,0)$ convolution layer, or in short: $conv(3,1,0)$.

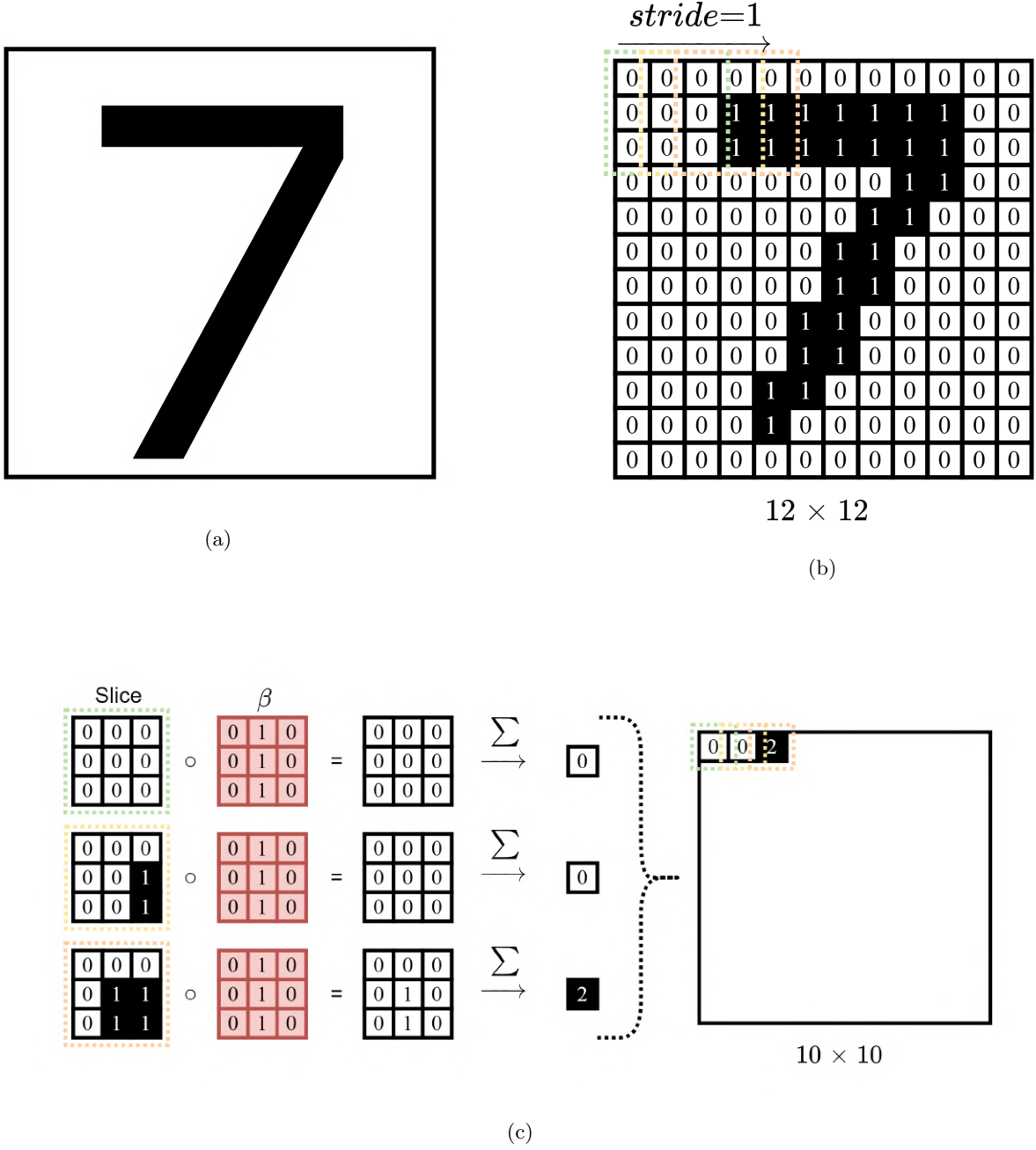


Figure 8: (a) Black and white image of a number ‘7’. (b) A one layer matrix representation (12×12) of this black and white image. A 0 indicates a white and a 1 indicates a black ‘pixel’. (c) A $(3,1,0)$ convolution filter with weights β (for simplicity only zeros and ones, but the weights can take any real number) strides across the image. For every image slice the Hadamard product with the filter is taken and the sum of all resulting elements is stored in the new matrix.

In Figure 8 the situation for convolution on a grey scale image is shown. However, coloured images have three input layers: red, green and blue (RGB). To deal with 3 input layers, the third dimension of a filter layer (the red layer in Figure 8c) needs to be adjusted to 3 accordingly. This means one filter will be of size $3 \times 3 \times 3$. One filter then gives three single values. These three values will again be summed, leading again to one value in the final output (see Figure 9). This means that, if we have an input image (RGB) of $400 \times 400 \times 3$ and we use a convolution layer with only one filter of $(3,1,1)$, the output of this layer will be one feature map of 400×400 . The number of filters determines the third dimension in the output after that convolution layer.

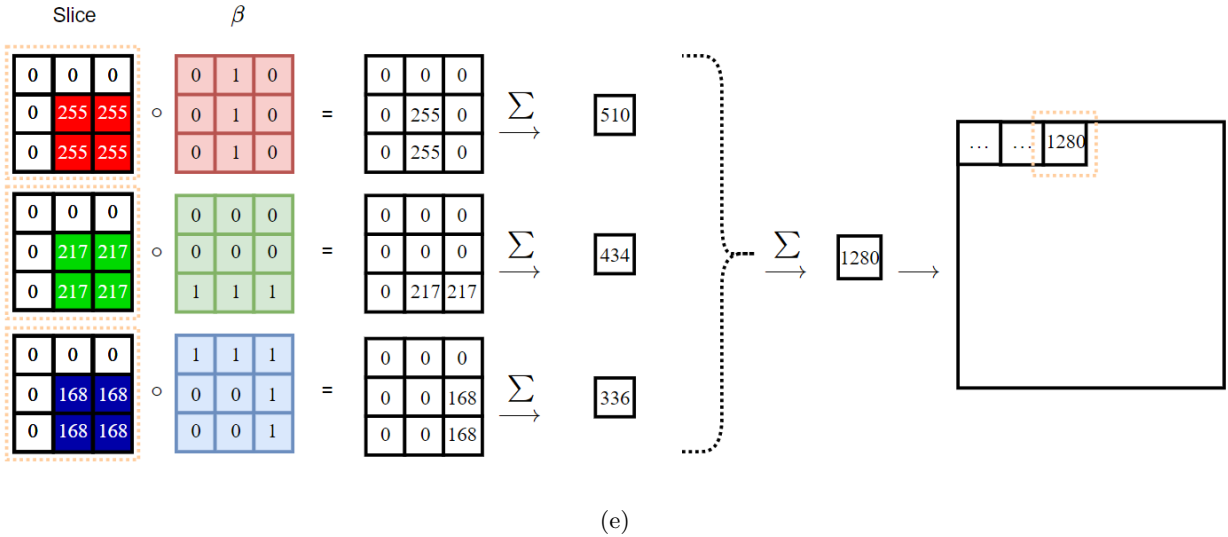
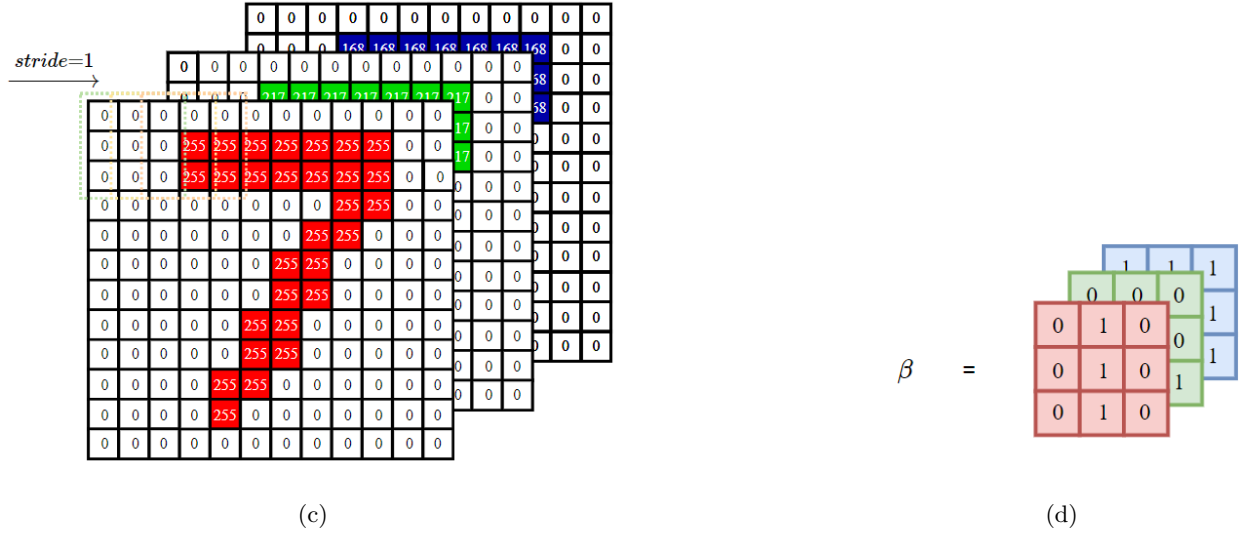
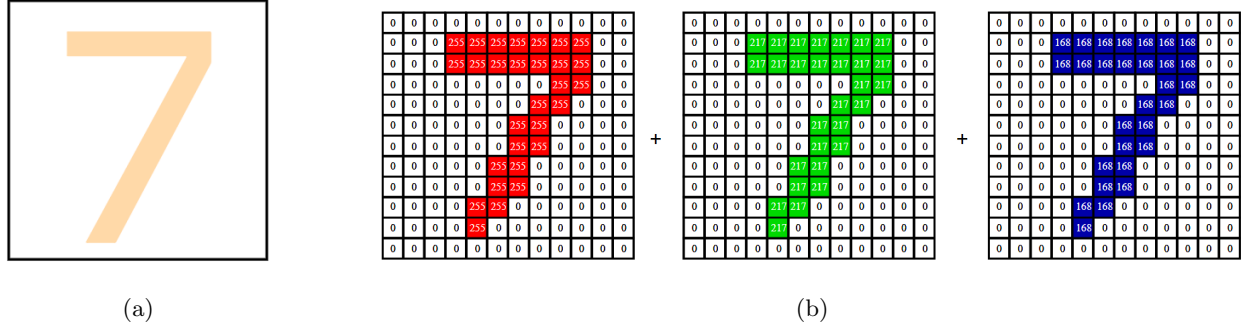


Figure 9: (a) Coloured image of a number ‘7’. (b) A three layer (RGB) matrix representation (12×12) of this image. A 0 indicates a white and a 1 indicates a black ‘pixel’. (c) Input to the CNN: $12 \times 12 \times 3$ (d) A (3,1,0) convolution filter with weights β , now this filter needs to have a third dimension with value 3 as well. (e) For only one stride location (important difference with Figure 8c!) the Hadamard product with the filter is shown and the sum of all resulting elements is stored in the new matrix.

This property can be used to control the number of feature layers at any point in the CNN. We can reduce or increase the number of feature layers, based on our needs. We will see this happening further on.

Pooling layers help to keep the amount of parameters to be trained small by reducing the size of the input. A 2×2 max pooling layer moves over an input image with a certain stride (e.g. 2), taking only the maximum value in that 2×2 block of the image and returning that value to the next layer. In this way the size of e.g. a 60×60 image is reduced to a 30×30 matrix. However, as a convolution layer is only interested in certain patterns, such a max-pooling layer will still keep patterns intact, only compressing this information into a smaller matrix. Normalisation layers help the network to be more robust to outliers, caused by e.g. the brightness of an image. A lot of bright images (with large RGB values) will influence the max-pooling and convolution layers, possibly making their weights overreacting during backward propagation on losses. This can cause the weights not to respond anymore when a much less bright image of a cat or dog is given as input. Normalisation can help in standardising the average pixel values in an image, therefore making the CNN less sensitive to outliers. In Figure 10 an example of a part of a CNN is shown with consequently a convolution layer, a max pooling layer and a normalisation layer.

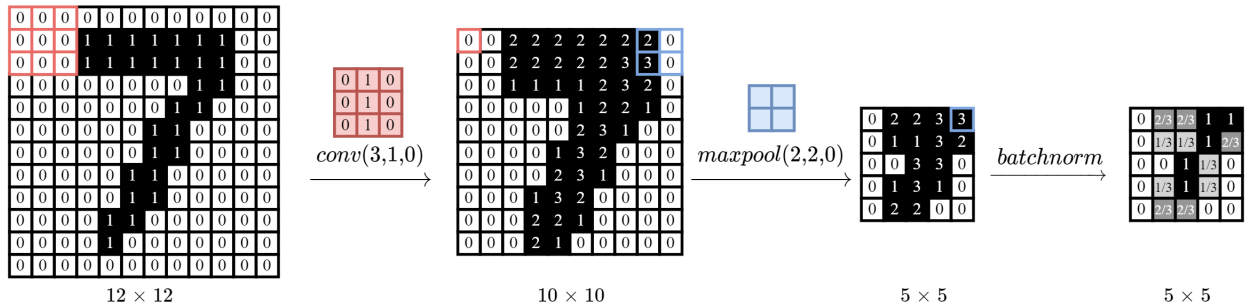


Figure 10: An implementation of consequently a convolution layer, a max pooling layer and a normalisation layer. Within the first two layers, the information is compressed from 144 cells to 25 cells.

Backpropagation in CNNs

As mentioned earlier, backpropagation is an important part of neural networks. This process of backpropagation works similarly for CNNs. As could be seen in Equation 10, δ_m is dependent on $h'(a_m)$. When adding more layers, this derivative term will recursively be included in the derivative chain, that is, for n extra layers the term is included n extra times when calculating the gradients in the first layer. The problem that arises with this property is that if the gradient is small in a few layers, this will result in little or no weight updating in the layers after these layer (when backpropagating from the last towards the first layer). This problem is called the ‘vanishing gradient problem’ and occurs when the derivative of the activation function becomes

close to zero for a certain range of values. E.g. the sigmoid function has a derivative close to zero for very large and small activation values. Using a large learning rate is not a sustainable solution, as this would make the model explode in layers where the gradient is not small. This is especially a problem for neural networks as the number of layers in these kind of networks increases rapidly if one wants to try to process images with it. Therefore, the smart construction of CNNs, that involves less layers than conventional (fully connected) neural networks, is already decreasing this vanishing gradient problem. The reason for this is the fact that there simply are less layers through which the gradient needs to be backpropagated, so also the vanishing gradient problem will occur less quickly. Besides this, in CNNs there are different ways how to solve this vanishing gradient problem, that might be still present despite the relatively small number of layers:

- Instead of a sigmoid activation layer on every layer, a lot of CNNs use ReLU activation functions. This activation function, described by $h(a) = \max(0, a)$ (Figure 11), has two advantages. First of all, it is able to output a much larger range of values than a sigmoid activation function (which generates outputs on the range $[0, 1]$). The ReLU activation function generates outputs on the range $[0, \rightarrow)$, which better represents how neurons in our brains work. In this way, very large inputs will also potentially generate large activation values, which indicates a better use of the information in x to predict y . A second advantage is that the derivative of the ReLU function is either 0 (when $a \leq 0$) or 1 (when $a > 0$) in the whole domain. This is especially important for solving the vanishing gradient problem; an error is either backpropagated or not, but the error is not slowly vanishing towards zero. This comes along with a disadvantage of using ReLUs, called the ‘dying ReLU problem’. As soon as a ReLU outputs zero, it means the gradient is also zero. When backpropagating, weights after this node will not be influenced anymore during gradient descent optimisation as the gradient of the ReLU is zero. This can cause whole parts of the neural network to ‘die’. Sigmoid or tanh activation functions (Figure 11) do also have the problem of very small gradients for extreme input values, but for these functions there is always still a small gradient allowing them to ‘recover’ and keep learning. A solution for this problem with ReLU activation functions is the so-called ‘leaky ReLU’, that artificially gives a very small gradient to negative activation values. A final advantage is that calculating the derivative of this function comes with very low computational costs (see the term $h'(a_m)$ in Equation 10).
- As mentioned above, problems especially arise for extreme input values, i.e. very small or very large inputs. Batch Normalisation can be used to normalise the input (demeaning and scaling of the standard deviation), which also causes the vanishing gradient problem to vanish or at least to be slowed down, making training faster.

Besides the measures above, another measure will be discussed in Section 4.3.

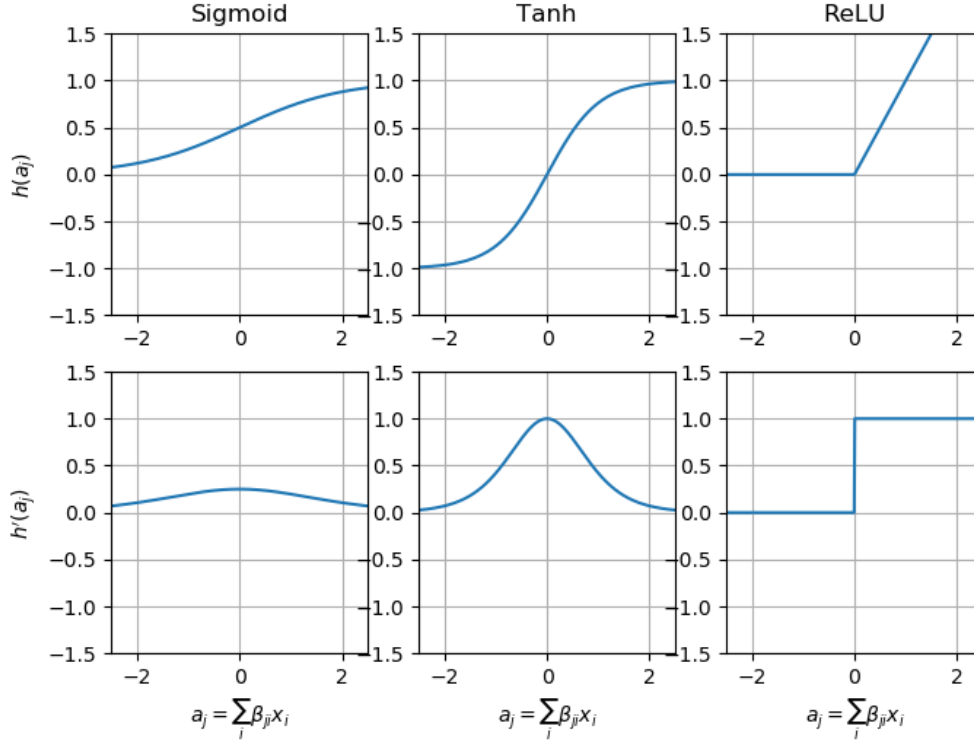


Figure 11: Comparison of three frequently used activation functions: sigmoid function, tanh function and ReLU function

A CNN called Residual Neural Network (ResNet) is the base of our Faster R-CNN model. However, we need to do object detection instead of image classification. Object detection comes along with the need to only examine specific parts of the image and see if there is an object. There can be multiple objects in one image, leading to two problems: we do not know how many objects are present in one image and we do not know what part of the image to extract to classify. In the following section we will address how we get these region proposals.

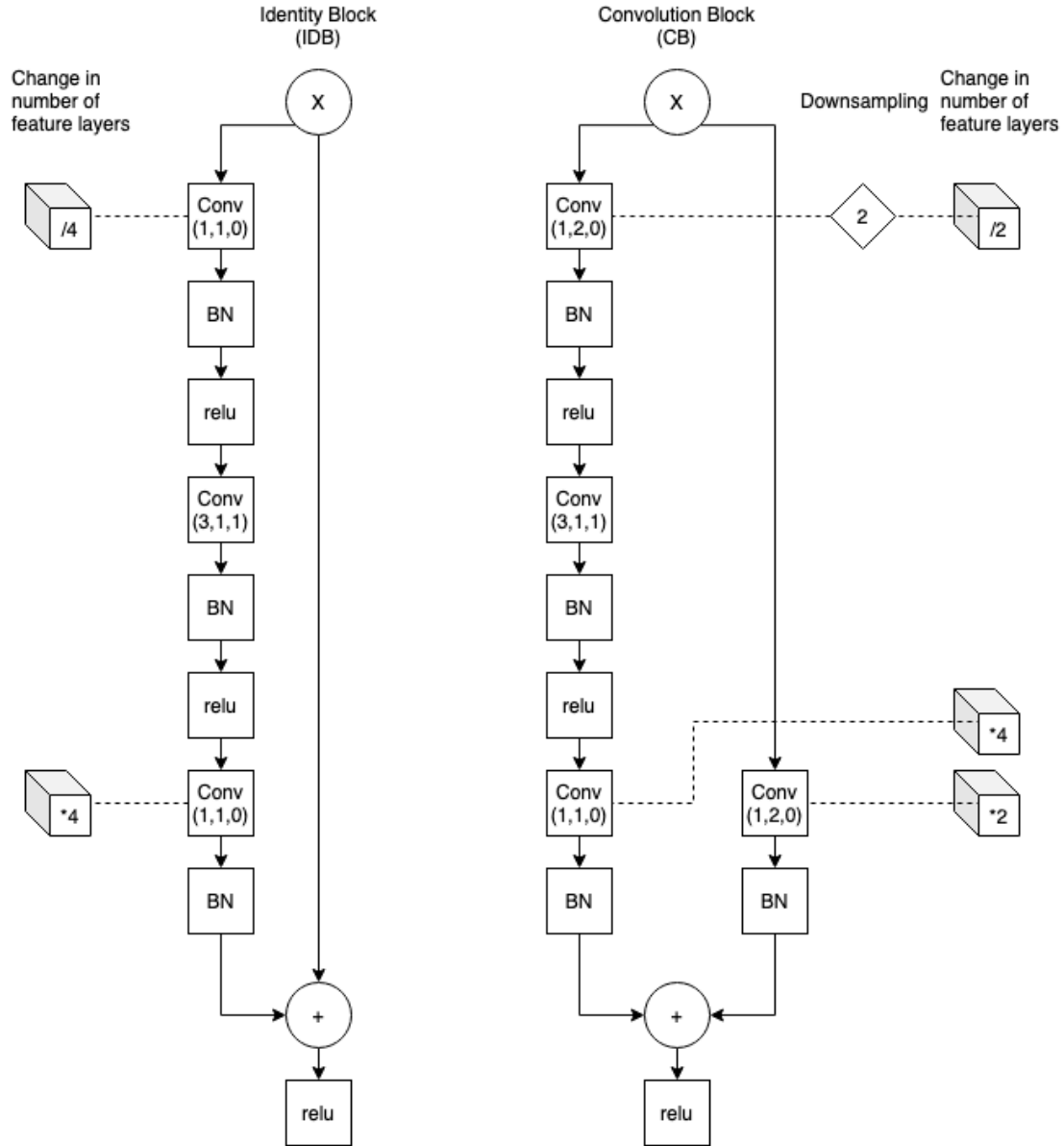
4.3 Faster R-CNN model


We start this section by examining the several building blocks a Faster R-CNN model is composed of. Then we specifically describe three important building blocks: the Region Proposal Network (RPN), the Non Maximum Suppression (NMS) layer and the Region of Interest Pooling (ROI) layer. Next, we examine the training of the model. We specifically explain the loss function and how anchors are used for training the RPN. Then we explain why the Faster R-CNN model cannot be trained end-to-end and how we solve this

using an alternating optimisation scheme, proposed by Ren et al. (2015). Finally we touch upon some input restrictions that are inherit to the way a Faster R-CNN works and we come up with some preprocessing steps that meet these restrictions.

The Faster R-CNN model consists of two parts. The first part is a Region Proposal Network (RPN), which is used to generate region proposals. Then these region proposals are fed into the second part, a Classifier (CLS), that classifies the proposed regions. In this research we use ResNet-50 as the base model of our Faster R-CNN model, in which ‘50’ stands for the number of convolution layers in the model.

The ResNet-50 model was originally developed as an image classifier (He et al., 2016). The model mainly consists of two building blocks that were specifically introduced with this model: identity blocks and convolution blocks. An identity block consists of convolution filters, batch normalisation layers (Ioffe and Szegedy, 2015) and ReLU activation layers and can be seen in Figure 12 on the left. As can be seen there, one path (on the right) is just passing the original input X forward, as the second path (on the left) tries to give extra information through functions based on this same input X . The idea behind this setup is that in this way a deeper network can easily perform at least as good as less deep networks, but it can potentially learn more by using the ‘residual’ (left) connection. In this way ResNet-50 also tackles the vanishing gradient problem; by backpropagation via the ‘shortcut’ the layers of the residual path are skipped. In this way the error backpropagates more easily towards the first layers of the network, making it easier for these layers to update their weights without quickly vanishing gradients. The convolution layers in the ‘residual’ path of the identity block have a kernel size (3,3), combined with a stride of (1,1) and (1,1) zero padding, resulting in an output which is as large as the input X , such that the addition of both matrices at the end of the block is possible. The (1,1,0) convolution layer at the start of an identity block reduces the amount of layers and the (1,1,0) convolution layer at the end increases the amount of layers again. The advantage of doing this is the amount of weight parameters needed compared to doing a (3,1,1) on all input layers directly. This latter option introduces around eight times as much parameters as the current implementation for each identity and convolution block. Reducing the depth with (1,1,0) convolution layers ensures that most information of these n original layers is still combined in the $\frac{n}{4}$ remaining layers. Convolution blocks are almost identical to identity blocks, except for the fact that also in the ‘shortcut’ path one convolution takes place. The setup of such a block can be found in Figure 12 on the right. These convolution blocks start with a convolution with a kernel size (1,1) and stride (2,2), resulting in a down-sampling with factor 2. The other convolution layers are the same as those in identity blocks, with kernel size (3,3) and a stride (1,1) with (1,1) zero padding. An exception is the first convolution block: here no down-sampling takes place, as this is already done in the first convolution layer that is present before the first convolution block (see Figure 13).



 = The first two dimensions of the output are reduced in size (downsampling), for example:
 $400 \times 400 \times 512 \rightarrow 200 \times 200 \times 512$


 = The third dimension of the output is reduced or increased in size, for example:
 $400 \times 400 \times 512 \rightarrow 400 \times 400 \times 256$

Figure 12: On the left side an identity block, consisting of Convolution layers, Batch normalisations and ReLUs. On the right side a convolution block. The first convolution block differs slightly, as the pooling layer before that block already did the downsampling. Instead of a (1,2,0) (K,S,P) the first convolution layer in the first convolution block is a (1,1,0) layer, resulting in no downsampling for the first CB and also a net increase in layers of a factor 4 instead of 2.

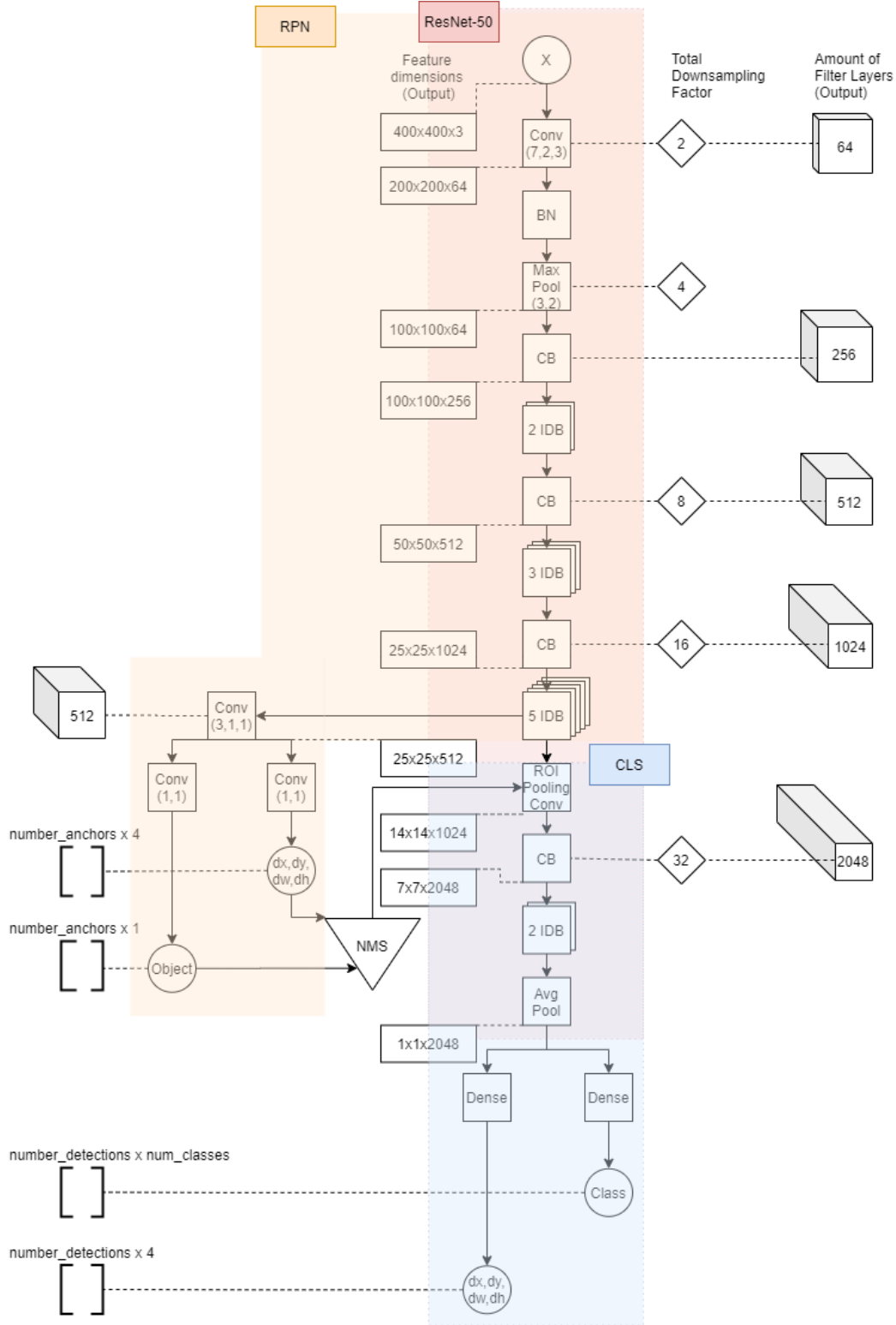


Figure 13: Faster R-CNN Architecture, developed by Ren et al. (2015). ResNet-50 (in red) is used as the base layers, upon which a RPN classifier and a R-CNN classifier are built. The complete RPN part of the model is shaded in yellow (approaching orange at overlap ResNet-50) and the complete CLS network is shaded in blue (approaching purple at overlap ResNet-50) In Figure 12 the architecture of CB (convolution block) and IDB (identity block) is specified.

The Region Proposal Network

The Region Proposal Network (RPN) is built upon the ResNet-50 network. After three sets of convolution and identity blocks (at a downsampling factor of 16), the features are passed into a (3,1,1) convolution layer, which uses the 1024 high-level layers from the ResNet architecture to extract features that determine whether an anchor is covering an object or background. Anchors are constructed as follows. As just said, the image matrix is downsampled with a factor 16 in the RPN. As can be seen in Figure 13, the original 400×400 matrix therefore is now of size 25×25 . Each of the elements in this 25×25 matrix represents a centre for a set of anchors. This set of anchors is defined by the user. In our research we use a set of anchors with the sizes $[16^2, 32^2, 64^2, 128^2, 256^2]$, where the unit is *pixel*². Using all these sizes, for each size we construct three *length : width* aspect ratios: $[1 : 1, 1 : \sqrt{2}, 2 : \sqrt{2}]$. These aspect ratios are chosen in such a way that the area of the set of the three smallest anchors is 16^2 for all three of them. This makes a total of 5×3 anchors per location and a total of $25 \times 25 \times 5 \times 3 = 9,375$ anchors for the whole image (as we had a matrix of size 25×25 , representing the whole image). This amount will be reduced as we ignore anchors that cross an image boundary. In Figure 14 a visualisation of this process is found for a smaller example, with an original image size of 12×12 pixels and a downsampling factor of 4, resulting in a matrix of 3×3 where the anchors positions are defined. Note that in this example the three larger anchors would in the end be disregarded as they fall partly outside the image.

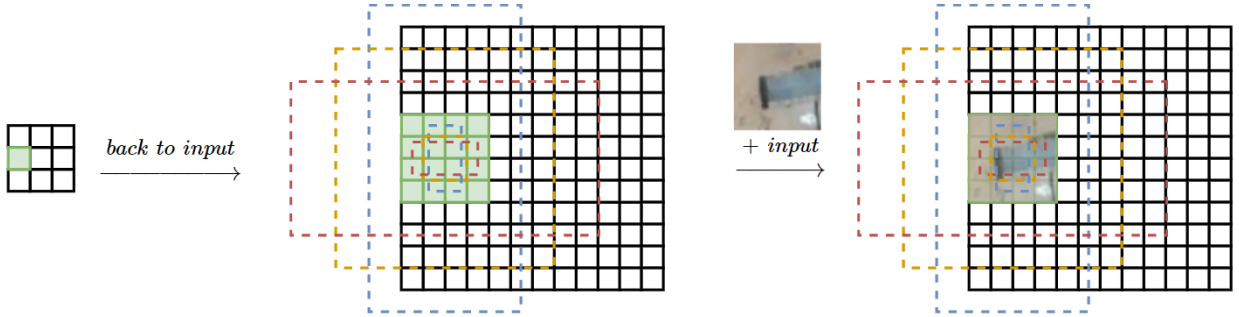


Figure 14: In the most left part of the image the 9 centres of the anchor positions are shown. We populate one anchor position (green) with 6 of the 15 anchors (middle part of the image) and relate the 4×4 pixels of the original image to an example of a part of the original input (right part of the image).

The (3,1,1) convolution layer returns 512 layers of features, which are flatten to one layer of the correct dimensions by the following two parallel connected (1,1) convolution layers (the notation (1,1) stands for a (1,1,0) convolution layer but in the case of zero padding the final 0 is often omitted). One of these layers generates an array with for each anchor the chance that it is covering an object and the second one generates a shift in location and size of the anchor, in such a way that it captures the object better.

Non Maximum Suppression

The RPN generates a lot of proposal regions. To decrease the computation time and to decrease to number of false positives that occur by multiple proposal regions covering the same object, Non Maximum Suppression (NMS). NMS starts by sorting all proposal regions by their chance of containing an object. Then, it picks the proposal with the best score and calculates the Intersect over Union (IoU) with all other proposals. All other proposals with an IoU larger than *overlap threshold* ($= 0.7$ in our research) are dismissed from further examination. This goes on till a maximum number of boxes *max boxes* ($= 300$ in our research) is picked. See Figure 15 for a visualisation of this process.

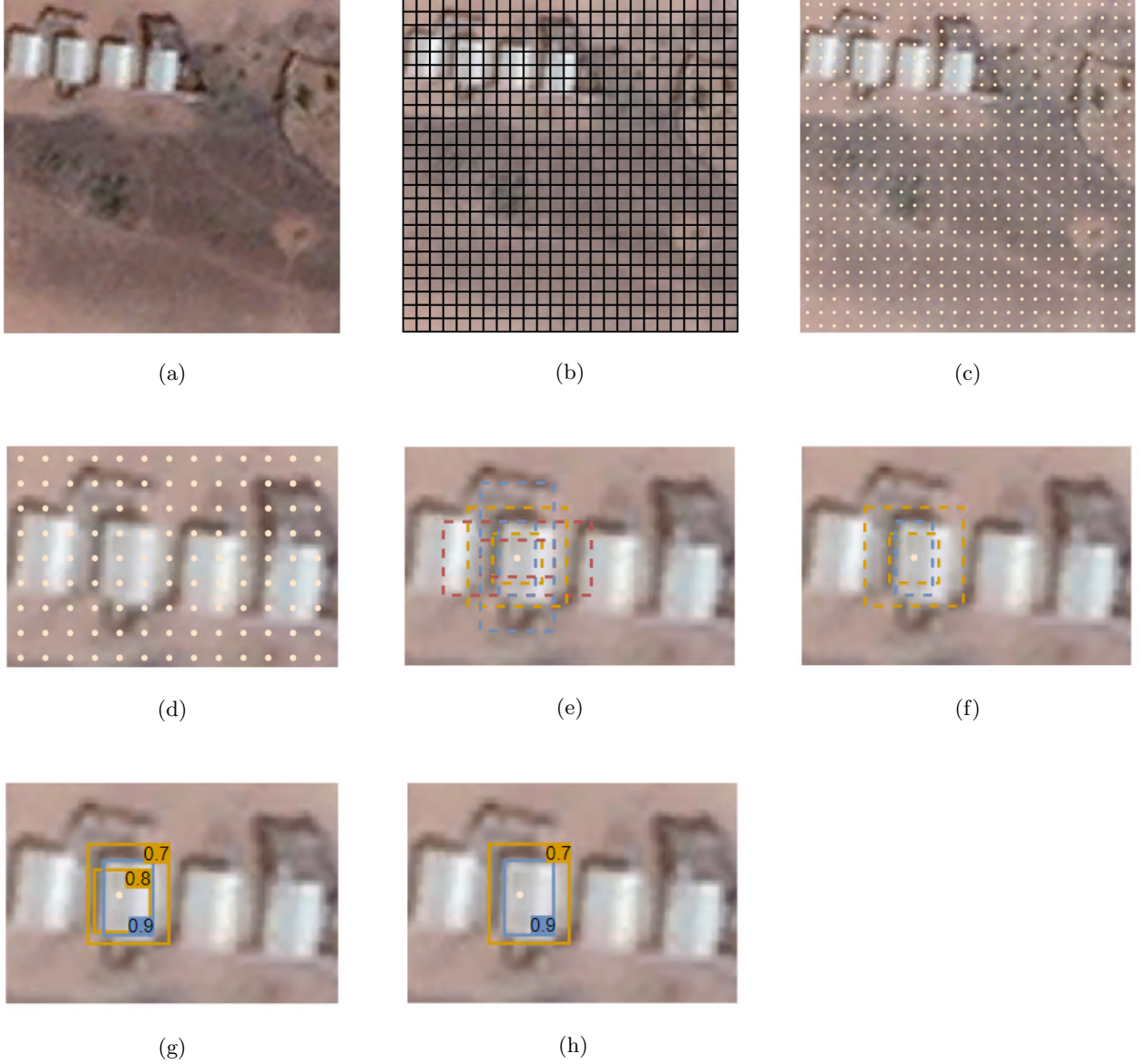


Figure 15: (a) An arbitrary image slice with four objects (white tents). (b) The same image slice with on it the 25×25 feature map matrix (c) The 25×25 anchor positions. (d) Zooming in on the location of the tents, we see that multiple anchor positions per tent are created. (e) Filtering only the anchors at one location, we have multiple candidates that overlap the ground-truth tent. (f) Filtering the most promising anchors, so that the visualisation is less messy. (g) The RPN predicts probabilities of the anchors containing a ground-truth object (\hat{p}). Also the RPN introduces small shifts (dx and dy) and small changes in aspect ratio (dw and dh) in the anchor locations, leading to better located bounding boxes. (h) Now the NMS layer filters proposals with an IoU larger than *overlap threshold* ($= 0.7$ during our research). The bounding boxes with $\hat{p} = 0.8$ and $\hat{p} = 0.9$ have an IoU larger than the threshold of 0.7 , so the bounding box with the lowest \hat{p} is dismissed from further examination.

Note that when looking at Figure 15h, we see that the larger bounding box, with a worse IoU with the ground-truth tent and therefore with a lower probability \hat{p} , is not filtered by the NMS layer. This can happen if the IoU of both boxes is lower than the threshold of 0.7. However, during testing a second NMS layer with a stricter (=lower) *overlap threshold* is added after the final output, that is, after the CLS network gives the final output. Also note that, although we only dismissed the other three anchors when going from Figure 15e to Figure 15f to get a clearer image, the NMS layer would probably have not filtered them as it will gather the 300 boxes with the largest probability of containing an object, \hat{p} . As we only have four tents in this image slice, all the anchors shown in Figure 15e are probably accurate enough to belong to the group of 300 best anchors. However, note that we also use anchors with a length of 256 pixels (more than half of the total image slice). These are not shown in Figure 15e, but those would most probably be filtered due to their low value for \hat{p} .

After NMS the generated proposal regions are fed into the classifier.

Region of Interest Pooling

The region proposals resulting from the NMS can differ in size for each proposal. As the classifier ends with two fully connected layers, the size of the input to the classifier should be standardised in some way. The Regions of Interest (ROI) layer solves this by subdividing the region of interest in a *pool size* \times *pool size* (= 14 \times 14) grid. The size of the region of interest does not have to be perfectly divisible by the number of pooling sections. In the original Faster R-CNN implementation He et al. (2016) extract the maximum value of each pooling section and pass these into the rest of the network, resulting in *num rois* (= 64) regions of interest of size *pool size* \times *pool size*. In this way a standard size of input is generated, which can be dealt with by the fully connected layers at the end of the classifier. However, in our implementation resize pooling (using bilinear interpolation) is used instead of max pooling, as this gives much faster compiling for the Tensorflow backend, while giving a similar performance, although some information is lost. An advantage of this method is the fact that region proposals that are smaller than *pool size* \times *pool size* can easily be dealt with.

After the final set of convolution and identity blocks, the features are passed into an Average Pooling Layer. This result is flattened to a vector, which can be used as input for the dense layers at the end of the network. One of these layers generates chances for a region to contain the possible objects that the network is trained on and the other layer again generates a refinement of the location of this bounding box.

Training

Normally neural networks are trained end-to-end. However, Faster R-CNN comes with some special properties. First of all, the network generates an output matrix for each of the four output layers, which requires the use of a multi-task loss function. Secondly, the output is generated at two locations in the network. The network can be seen as two networks, one RPN part of the model and one classifier network, that share some layers but also have some unique layers. Together with the fact that the classifier network receives input at the ROI Pooling layer, these two properties give rise for the need of a different training method.

Loss function

The multi-task loss function for the RPN consists of two parts: a part where the loss is calculated for the predicted classification (background/object) and a part where the loss is calculated for the location of the bounding box, relative to the anchor. That is, a regression is done on the location and width and height of the proposed anchor with respect to the ground truth location of the object most nearby. To make this learning of locations of an object scale invariant, the following transformation (Ross Girshick et al., 2014) is used. Anchor A_i and ground-truth bounding box G_i have a centre coordinate and width and height, denoted with respectively the subscripts x, y, w, h . The matrix with pool features contained by the anchor are denoted with $\mathbf{A}_{f,i}$ and the linear relationship between features $\mathbf{A}_{f,i}$ and the ground-truth bounding box are denoted by $\hat{d}_*(\mathbf{A}_{f,i})$, where $*$ can be x, y, w, h . The hat indicates that the parameter vector that $d_*(\cdot)$ consists of, is estimated, resulting in estimations of the ground-truth bounding box G_i . Then the transformation

$$\begin{aligned}\hat{G}_{x,i} &= A_{w,i} \hat{d}_x(\mathbf{A}_{f,i}) + A_{x,i}, \\ \hat{G}_{y,i} &= A_{h,i} \hat{d}_y(\mathbf{A}_{f,i}) + A_{y,i}, \\ \hat{G}_{w,i} &= A_{w,i} \exp(\hat{d}_w(\mathbf{A}_{f,i})), \\ \hat{G}_{h,i} &= A_{h,i} \exp(\hat{d}_h(\mathbf{A}_{f,i})),\end{aligned}\tag{12}$$

is scale invariant due to the scaling term directly after the equation sign in the first two equations. Log-space translations in the third and fourth equations are used to prevent \hat{G}_w and \hat{G}_h from becoming negative. These equations can be rewritten as the following estimation models:

$$\begin{aligned}\hat{t}_{x,i} &= \frac{\hat{G}_{x,i} - A_{x,i}}{A_{w,i}} = \hat{d}_x(\mathbf{A}_{f,i}), \\ \hat{t}_{y,i} &= \frac{\hat{G}_{y,i} - A_{y,i}}{A_{h,i}} = \hat{d}_y(\mathbf{A}_{f,i}), \\ \hat{t}_{w,i} &= \log\left(\frac{\hat{G}_{w,i}}{A_{w,i}}\right) = \hat{d}_w(\mathbf{A}_{f,i}), \\ \hat{t}_{h,i} &= \log\left(\frac{\hat{G}_{h,i}}{A_{h,i}}\right) = \hat{d}_h(\mathbf{A}_{f,i}),\end{aligned}\tag{13}$$

which are based on the assumed DGPs:

$$\begin{aligned}
t_{x,i} &= \frac{G_{x,i} - A_{x,i}}{A_{w,i}} = d_x(\mathbf{A}_{f,i}) + \epsilon_{x,i}, \\
t_{y,i} &= \frac{G_{y,i} - A_{y,i}}{A_{h,i}} = d_y(\mathbf{A}_{f,i}) + \epsilon_{y,i}, \\
t_{w,i} &= \log\left(\frac{G_{w,i}}{A_{w,i}}\right) = d_w(\mathbf{A}_{f,i}) + \epsilon_{w,i}, \\
t_{h,i} &= \log\left(\frac{G_{h,i}}{A_{h,i}}\right) = d_h(\mathbf{A}_{f,i}) + \epsilon_{h,i}.
\end{aligned} \tag{14}$$

As $d_*(A_{f,i}) = \mathbf{A}_{f,i} \mathbf{w}_*$, these equations boil down to linear equations that can be solved with regression. A Huber loss function, as proposed by Ross Girshick (2015), is used as this makes the learning process less sensitive to outliers. The loss function

$$L_{loc}(e_i) = \begin{cases} \frac{1}{2}e_i^2 & \text{for } |e_i| \leq 1, \\ (|e_i| - \frac{1}{2}) & \text{otherwise,} \end{cases} \tag{15}$$

describes the loss for observation i . As we have 4 equations for each observation i (see Equation 14), we replace e_i in Equation 15 with $e_{*,i}$, where $e_{*,i} = t_{*,i} - \hat{t}_{*,i}$. This leads to the total loss

$$L_{reg}(t_i, \hat{t}_i) = \sum_{i=1}^N \sum_{j \in \{x,y,w,h\}} L_{loc}(t_{j,i} - \hat{t}_{j,i}), \tag{16}$$

where N is the number of positive- and negative-labelled anchors. t_i is the 4×1 vector containing the observations $t_{x,i}$, $t_{y,i}$, $t_{w,i}$ and $t_{h,i}$ while \hat{t}_i is also a 4×1 vector, containing the predictions of these four measures. During training, the following set of rules is used to label anchors as positive or negative, each label based on a combination of one anchor with one ground-truth bounding box:

- The anchor with the highest IoU with a ground-truth box is matched to that ground-truth box and labelled as a positive;
- An anchor with an IoU overlap larger than 0.7 with any ground-truth box is labelled positive;
- An anchor with an IoU overlap smaller than 0.3 with all ground-truth boxes is labelled negative.

This means one ground-truth box can assign more than one positive label to anchors. Anchors that are not labelled positive or negative, are not used for training the RPN part of the model. $\hat{t}_{j,i}$ is the outcome of the RPN convolution layer (most left output in Figure 13), which compresses the 512 layers to four values per anchor through (1,1) convolution in combination with a linear activation function.

The other part of the loss function L_{cls} consists of a log-loss function based on the classification of the box (positive or negative) and the predicted chance of an anchor box to contain a ground-truth object:

$$L_{cls}(p_i, \hat{p}_i) = - \sum_{i=1}^{N_{cls}} p_i \log \hat{p}_i. \tag{17}$$

This binary cross-entropy loss function obtains \hat{p}_i from the second output of the RPN part of the model, which is also a (1,1) convolution layer. This layer generates one value through a sigmoid activation function. As the RPN has a lot of input anchors, a bias towards predicting all anchors as background may be created when using all positive and negative anchors. To avoid this, a mini-batch of 256 anchors (Ren et al., 2015) is used for training the RPN, using up to 128 randomly positive-labelled anchors and filling up this mini-batch with randomly chosen negative-labelled anchors. This ensures $N_{cls} = 256$ and this also limits the value of N (Equation 16) to an upper boundary of 256.

In Figure 16 an example can be found of this matching algorithm between a ground-truth box and multiple anchor proposals and how this leads to loss terms to learn from by Equation 16 and Equation 17. This example follows on Figure 14. We have selected the two anchors from Figure 14 that are closest to the ground-truth object, a refugee tent. As we just said, the anchor with the highest IoU with a ground-truth box is matched to that ground-truth box and labelled as a positive; we see this happening in the left part of the figure. At the original anchor position, we denote a '1' for y_{cls} , indicating an object. All other values in this 3×3 matrix will be either 0 or 1, depending on the contents of the rest of the original image. Furthermore, a small shift to the right is needed to let the anchor box fully coincide with the ground-truth object; we denote $dx = 0.55$, which is one of the terms (namely $t_{x,i} - \hat{t}_{x,i}$, for image i) that will be used to calculate the regression loss (Equation 16). However, we will not use the yellow anchor on the right side of the image, as it does not satisfy one of the three requirements just described: the anchor does not have the highest IoU with a ground-truth box, it does not have an IoU overlap larger than 0.7 with any ground-truth object and it does have an IoU overlap smaller than 0.3 with all ground-truth objects. Going back to our own network, when we do this for all anchor configurations (combination of size and aspect ratio) we obtain 15 matrices of size 25×25 for y_{cls} , indicating positive anchors for each anchor configuration. We also do this for the four output matrices per anchor configuration describing y_{rpn} , so this results in 60 matrices of size 25×25 . However, one addition should be made on this. When we consider the third of the three requirements for which an anchor is labelled (an anchor with an IoU overlap smaller than 0.3 with all ground-truth boxes is labelled negative), we need to store these negative labels in separate 3×3 matrices, so the model does not confuse positive with negative labels. Therefore we end up with twice as much 25×25 matrices: for y_{cls} we have 30 matrices and for y_{rpn} we have 120 matrices to learn from. Because of this way of constructing the training, we can use the (3,1,1) convolution layers separately. To account for the varying anchor sizes and aspect ratios, each regressor (feature map) is responsible for one combination of scale and aspect ratio. The regressors do not share weights, which makes it still possible to predict boxes of various sizes even though the feature layers are of a fixed size/scale (they are all (3,1,1) convolution layers), thanks to the design of anchors (Ren et al., 2015).

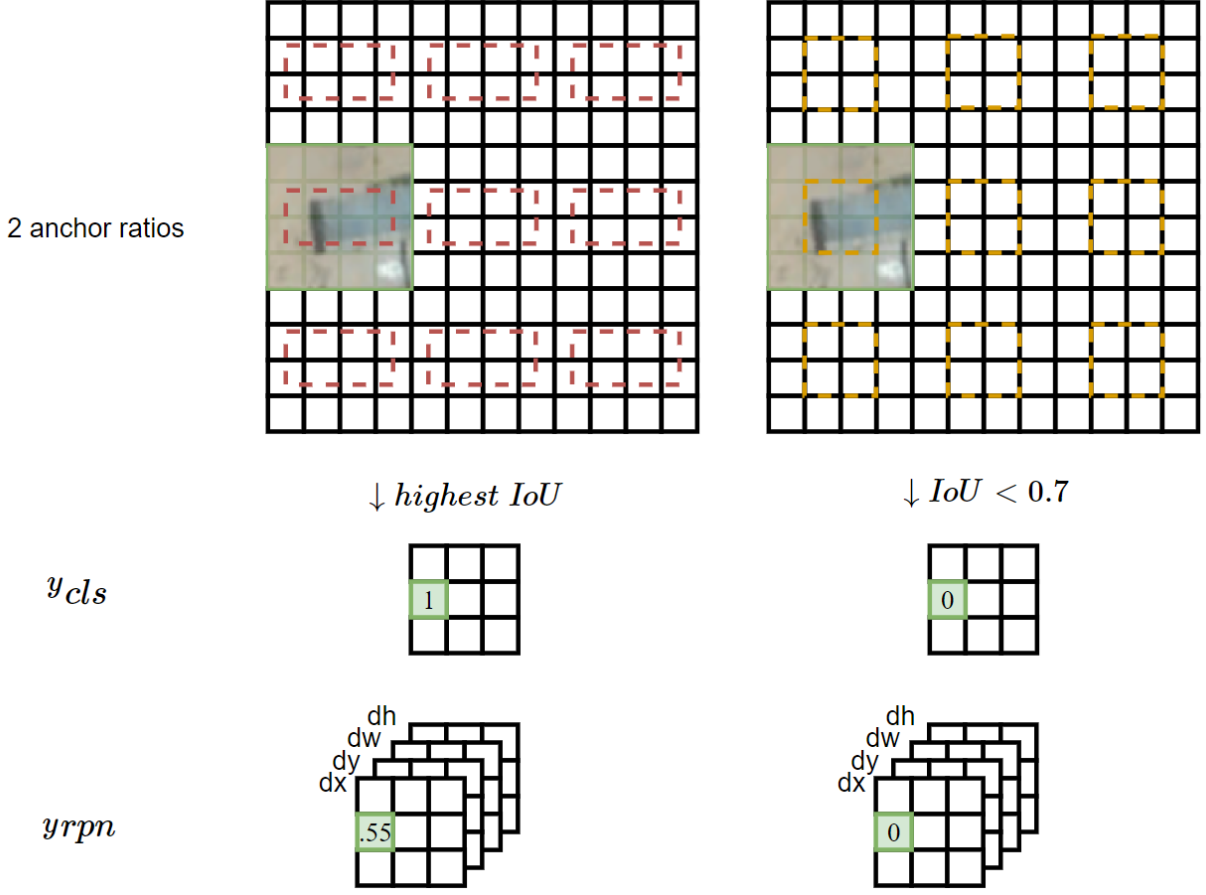


Figure 16: Selection of positive labelled anchors, resulting in matrices with information on p_i and \hat{p}_i (through y_{cls}) and matrices with information on $t_{j,i}$ and $\hat{t}_{j,i}$ (through y_{rpn}).

Both loss functions from Equations 16 and 17 can be combined in one loss function for the RPN part of the model

$$L(p_i, \hat{p}_i, t_i, \hat{t}_i) = \frac{1}{N_{cls}} \sum_{i=1}^{N_{cls}} L_{cls}(p_i, \hat{p}_i) + \lambda \frac{1}{N_{reg}} \sum_{i=1}^N p_i L_{reg}(t_i, \hat{t}_i). \quad (18)$$

which implies the second part of the equation is only activated for positive labels. As Ren et al. (2015) mention that N_{reg} , the amount of anchor locations ($N_{reg} = W \times H \approx 2,400$ for a convolution feature map with downsampling factor 16), is compensated by choosing $\lambda = 10$, such that $\frac{1}{N_{cls}} \approx \lambda \frac{1}{N_{reg}}$ and both losses are weighted approximately equally, we implement the simplified form

$$L(p_i, \hat{p}_i, t_i, \hat{t}_i) = \frac{1}{N_{pos}} \sum_{i=1}^{N_{cls}} L_{cls}(p_i, \hat{p}_i) + \lambda \frac{1}{N_{pos}} \sum_{i=1}^N p_i L_{reg}(t_i, \hat{t}_i). \quad (19)$$

with $\lambda = 1$ and N_{pos} the amount of positive-labelled anchors, with $p_i = 0$ for negative labels and $p_i = 1$ for positive labels.

After one batch training of RPN, the same RPN predicts proposal boxes for all the original anchors, based on the new weights. After the NMS-layer the selected number of boxes ($\leq \text{max boxes}$) are used to train the classifier network. The proposal boxes that come out of this NMS step are matched to the ground truth anchor with the largest IoU. The proposal boxes are then subdivided into three categories:

- Negative samples: if for a box the best IoU is smaller than *classifier min overlap*, they are discarded, as little can be learnt from these proposals;
- Hard negative samples: if for a box holds that *classifier min overlap* < the best IoU < *classifier max overlap*, this proposal is regarded as a hard negative;
- Positive samples: if the best IoU is larger than *classifier max overlap*, this is regarded as a positive sample.

After discarding the negative samples, a number of randomly chosen proposal regions, called Regions of Interest (ROI), is passed as input to train the classifier. We denote this number of randomly chosen ROIs with *num rois*. These ROIs are passed into the classifier with a ratio positive samples:negative samples of up to 1:1. That is, if more than $\frac{\text{num rois}}{2}$ positive samples are available, then half of the randomly chosen proposal regions consists of positive samples and half of it consists of negative samples. However, if less than $\frac{\text{num rois}}{2}$ positive samples are available, all these positive samples are chosen after which the rest of the needed proposal regions is taken from the negative samples, till we again have a total number of regions equal to *num rois*. This sampling from the original *max boxes* number of proposal regions is done as in most cases the amount of negative samples will dominate this set, causing a bias in the classifier towards predicting ‘background’.

For both outputs of the classifier network the same loss functions are formulated as for the RPN part of the model, with some little modifications. For the classifier output loss with K classes, a K –categorical cross-entropy function is used, in which the background class ($K = 0$) is excluded again:

$$L_{K,cls}(p_{i,k}, \hat{p}_{i,k}) = - \sum_{i=1}^{N_{cls}} \sum_{k=1}^K p_{i,k} \log \hat{p}_{i,k}, \quad (20)$$

Alternating optimisation scheme

As said earlier, end-to-end training of the full network is not straightforward due to the input of region proposals halfway of the network. This input of ROIs halfway of the network was constant in Fast R-CNN as it was generated by an external selective search algorithm, which made it possible to train this network end-to-end with back-propagation, as $\frac{\partial L}{\partial ROI_j} = 0$, where $j \in \{x, y, w, h\}$. However, in the Faster R-CNN model the ROI input is dependent on the convolution feature map created earlier in the network, causing

$\frac{\partial L}{\partial ROI_j} \neq 0$ in general and the ROI Convolution Pooling layer is not differentiable with respect to ROI. A theoretically exact solution is therefore not straightforward and unlike earlier methods like Fast R-CNN and R-CNN, this network is trained iteratively, starting with the RPN part of the model. The following alternating optimisation scheme is proposed by Ren et al. (2015):

1. Train the RPN part of the model from the base model with weights W_0 , to get weights W_1
2. Generate proposal regions P_1 using the RPN with weights W_1
3. Train the classifier, starting with weights W_0 and proposals P_1 , to obtain weights W_2
4. Train the unique layers of the RPN part of the model using weights W_2 to obtain weights W_3 , while freezing the shared base layers
5. Generate proposal regions P_2 using the RPN with weights W_3
6. Train the unique layers of the classifier, starting with weights W_3 and proposals P_2 , to obtain weights W_4 . Again the base layer weights are frozen
7. Combine the updated weights W_3 from the RPN layers with the updated weights W_4 from the classifier layers to get the final model.

Although this alternating optimisation is not based on fundamental principles, Girshick (2015) found out this method works, because the RPN part of the model, that generates the non-differentiable input, receives direct supervision by optimising the loss function from Equation 19. It turns out that although error propagation from ROI pooling might help, it is not strictly needed (Girshick, 2015). However, we do not implement the training scheme like this but we only implement steps 1, 2 and 3. The two reasons for this choice are that it takes a lot of extra training time to include also steps 4 to 7, also involving extra hyperparameters (e.g. the ratio training epochs in steps 1 and 3 to steps 4 and 6). Also to compare different preprocessing steps and model adaptations we do not need this complete alternating scheme. Every mini-batch consists of one image, with the amount of mini-batches per epoch equal to *epoch length*. We use an *epoch length* of 1,000 images during our whole research. The number of epochs is defined by *epoch num*.

In order for the model to be able to detect buildings in general we pre-train the model on the labelled dataset of the SpaceNet Challenge. We take the satellite images from the Rio de Janeiro set and convert the polygons to bounding boxes. We start with ResNet-50 weights trained on the ImageNet dataset and the outcome of this pre-training can be used as W_0 . The size of the bounding boxes in the SpaceNet dataset is determined and all bounding boxes with an area smaller than 140 are filtered out of the dataset, for three reasons. First of all, these buildings on the satellite images are almost not visible by eye. Secondly, such

small buildings do also not occur in our dataset and finally such small objects are not detectable for an RPN with downsampling factor 16. Theoretically, Eggert et al. (2017) derive that the minimal object size that can still be detected in the worst case (i.e. when the ground-truth bounding box starts exactly halfway of the convolution grid with downsampling factor d , so at $\frac{1}{2}d$ pixels from an anchor in the original image), is formulated by

$$s_g \geq \frac{d(t+1) + d\sqrt{2t(t+1)}}{2-2t}, \quad (21)$$

in which t the *classifier max overlap*. This parameter represents, as explained earlier, the minimum overlap of a proposal box with a ground-truth bounding box above which the classifier uses a proposal box as a positive box during training. Here, d is the sub-sampling factor in the RPN part of the model, which is 16, as explained earlier; s_g is the minimal object size in pixels that can still be detected. Using *classifier max overlap* = 0.5 as is done by Ren et al. (2015) and $d = 16$ we end up with $s_g \geq 44$.

After pre-training on the SpaceNet dataset, we try direct testing on the camp data and we try training the model some more on the camp data before testing it. The *Range* parameter of the Google Images varies from 100m up to around 1900m, giving all kind of sizes of tents. Pre-training on the SpaceNet will make clear what range of bounding box sizes are detected correctly and if necessary, we can adapt the scales of the Google Images accordingly. According to Equation 21 $s_g \geq 44$, which means that bounding boxes should have at least one side, but preferably both sides with 44 pixels or more.

Beside this restriction, we also want to restrict the size of the input images as a larger size comes along with more anchors and more positions of the convolution layers to check. Too large input might give memory issues. We choose the images to be of size 400×400 pixels. As a consequence, we need to slice our 3140×1980 images to this size. A problem that might occur now is that due to the different *Range* levels some sliced might completely be filled with an object. Worse, if one object is spread out across multiple slices, this makes it very hard to be recognised by the convolution filters. To solve this, we also introduce a maximum size of the bounding boxes. Using three images from the training set we conclude that the *Range* parameter has a linear relationship with the amount of meter that is represented by one pixel, by the formula

$$meter/pixel = 5.97 \cdot 10^{-5} + 3.67 \cdot 10^{-4} \cdot Range, \quad (22)$$

As these tents are typically in the range of 7×7 m up to 35×35 m, we resize the original images such that 7×7 m tents are never too small and 35×35 m tents are never too large. The former criterion leads to a upper bound of 0.25 meter/pixel, while the latter criterion suggests an lower bound of 0.10 meter/pixel. Images out of this interval are resized so their resolution matches the nearest interval bound.

4.4 Boosting

We start this section with assessing the mAP scores that never exceed 70%. We pose a hypothesis for this limited accuracy score and propose to use a boosting method to solve this problem. Next, we examine an existing boosting method called AdaBoost and we discuss how we can use this algorithm when training a Faster R-CNN. We end this section with some implementation details of the boosting method.

As we will see in Section 5, most results are given after only 10 or 20 training epochs, where one epoch consists of training on 1,000 image slices. During training we discovered that after this number of epochs the performance measures of the model decreased and overfitting on the training set was already happening. The fact that setting the parameters such that training is done rather quickly is not necessarily a bad thing. However in best cases the mAP never reached scores above 70%, which means there is still room for improvement. A reason for this could be that our training set is too small. With 6,886 image slices at most we expect every image to pass the algorithm at least once when training 20 epochs (=20,000 images). Although this sounds as quite a large image set, note that only half of these 6,886 image slices have tents on it. These 3,443 image slices originate from 78 satellite images of 11 distinct refugee camps. So a lot of tents will look quite similar in several slices, although the introduced contrast stretching (see Section 5) per slice will also introduce more variance between these slices. Another reason for the fact that the mAP score is never exceeding 70% could be that the model is learning too easily. The white tents do not have very ‘deep’ features. That is, compared to classifying a taxicab, hammerhead shark or an oxygen mask (all classes in the ImageNet dataset from which we use the pretrained weights), all we have to detect are white tents, which are basically sets of white pixels grouped in rectangles. Therefore, one would expect the model to obtain a rather high performance quite easily, as current state of the art models reach scores over 85% on the ImageNet dataset. A reason why performance scores above 70% are not obtained for our model, could be that this relatively easy learning task leads to arriving in a local minimum rather quickly. Especially combined with the fact that the training set is not that large, the training could become biased in the direction of the camps with the largest number of tents, as those tents will dominate the training set in numbers and also look really similarly (same angle of the sun in one image, same set up of the tents, same spacing, same orientation). Although this is partly countered by data preprocessing in the training set (e.g. the contrast stretching, turn images by 90 or 180 degrees, flip images), we should consider the fact that CNNs are often very deep learning algorithms, leading to easier overfitting when the training set is too small. To prevent the model from being biased towards images with a relatively large number of tents, we try using boosting to focus on the errors made during training, as these errors will be larger for images with smaller numbers of tents.

In general boosting methods try learning from weak learners by giving more weight to data points that

have larger training errors. In the next iteration this will influence the loss function as the next learner will put more emphasis on fitting correctly to the data points that were not correctly fitted to in the previous iteration. We will try using boosting to improve the performance of the Faster R-CNN model. The idea is that by doing this, each learner may be able to find a different local minimum for the data used by the learner. As stated before, arriving rather easily in a local minimum due to a rather shallow feature depth could be problematic. By using some kind of combination of weaker learners and combining them with majority voting we might be able to detect more tents. In order to create a weak learner, we only take the RPN part of the Faster R-CNN model. In the following we will focus on the original two-class classification algorithm of AdaBoost, as we only deal with the classes ‘background’ and ‘white tent’. This AdaBoost model is the base model we want to implement in the Faster R-CNN model. However, Hastie et al. (2009) have extended the AdaBoost algorithm to a multi-class case. The final classifier of AdaBoost is as follows:

$$K(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right), \quad (23)$$

where T is the number of weak classifiers, $h_t(x)$ is the output of classifier t and α_t is the importance of classifier t , based on how good the performance of classifier t is. As $h_t(x)$ is limited to -1 or $+1$ in the original paper, this function boils down to a majority vote. Originally, the AdaBoost algorithm works as follows:

Algorithm 1 AdaBoost; training on N data points, where $D_{t,i}$ is the error weight of data point i for learner t , α_t the importance of learner t , reflecting how well the learner performs on the train data and $I(y_i \neq h_t(x_i))$ is an indicator function returning 1 if $y_i \neq h_t(x_i)$

- 1: **for** i from 1 to N **do** $D_{0,i} = \frac{1}{N}$
 - 2: **for** t from 1 to T **do**
 - 3: Train the model (often a decision stump) on all N data points simultaneously to minimise the weighted fraction of misclassified training data points e_t with $e_t = \frac{\sum_{i=1}^N D_{t-1,i} I(y_i \neq h_t(x_i))}{\sum_{i=1}^N D_{t-1,i}}$
 - 4: Calculate the importance $\alpha_t = \frac{1}{2} \log \left(\frac{1-e_t}{e_t} \right)$
 - 5: **for** i from 1 to N **do**
 - 6: Calculate the new error weights for the next iteration $D_{t,i} = \frac{D_{t-1,i} \exp(-\alpha_t y_i h_t(x_i))}{\sum_{i=1}^N D_{t-1,i} \exp(-\alpha_t y_i h_t(x_i))}$
-

Two challenges arise when implementing this algorithm in a Faster R-CNN model. First of all we need some kind of ‘weak learner’. Secondly, AdaBoost requires that this weak learner should be better than random guessing. In a classification problem this is a clear requirement as the loss function generally consist of only one part, namely the predicted class compared to the real class. However, for object detection our loss function consists of four parts: both a classification and regression loss for both the RPN and the CLS part

of the model. Especially the regression loss, which contains errors on the location and size of the estimated bounding box, is not easily expressed in terms of a random guess. To assess the first problem, we propose using only the RPN part as weak learner. This part only indicates whether there is an object or not in the anchor box and is therefore considered a ‘weak learner’ compared to the whole model. Also, as we are only interested in finding one class, namely ‘white tents’, we have done some tests only using the RPN part of the model to find these tents, but indeed the RPN part of the model alone seemed too weak to correctly classify the white tents correctly. For this reason we will use AdaBoost only on the RPN part of the model and we will train the CLS network without AdaBoost. The exact implementation is described later in this section. To assess the second problem, we will combine the classification and regression loss from the RPN and use the combined loss as our loss function (Equation 19). Note that this score still does not reflect an easy measure to be benchmarked against a random guess. However as an object is only classified as such when the network gives a probability $\hat{p}_i > 0.5$, the classifier part of the loss function can be seen as ‘random guess’. To summarise, we solve the challenge of using a weak learner by only boosting the RPN part of the model and we use a combined loss term from both outputs of the RPN to get a useful error measure.

Besides this, in a lot of applications of AdaBoost training is done on the whole training set at once (e.g. when applying AdaBoost on decision stumps). However, Faster R-CNN trains on one image at the same time. It does not make sense to update the error weights based on one data point because under the mentioned normalisation we cannot update the weight of only one data point. Therefore the algorithm has to be designed differently. The other extreme would be updating the error weights after training once on the whole data set. But when training on the whole dataset we make 6,886 training iterations before we can update the error weights. So a compromise between these boundary cases is needed. As the training of our model without AdaBoost is already split up in ‘epochs’ of 1,000 randomly chosen image slices from the train data set, we will use this epoch size as well to update the error weights. This is also a convenient choice when comparing performance of the model using AdaBoost with the base case model. This means every learner is weak in another sense, i.e. that it is only being trained on a subset of the train data. As said before, normally every weak learner is being trained on the whole train set. Although we only pick a subset of the images to train on every iteration, we want the model to be able to focus more on image slices with a larger ‘error’ (i.e. the loss L_i , Equation 19). To increase the chances of image slices with a large error to occur in the next learner as well, we propose using the error weights as representation of the probability the image slice will be used during the next epoch. Because we allow sampling with replacement, this means one epoch of 1,000 images could contain one train image slice multiple times, depending on chance of picking image slice i during epoch t $p_{i,t} = e_{i,t}$.

The next adjustment to Algorithm 1 is in calculating the importance of each ‘stump’. In the AdaBoost originally the following formula is being used: $\alpha_t = \frac{1}{2} \log \left(\frac{1-e_t}{e_t} \right)$. However, in our application two problems occur with this classical definition. Firstly, this formula assumes that e_t only adopts values on the interval $(0, 1]$. This is true for the classical applications of AdaBoost in classifier decision stumps, where the error term e_t is the proportion of misclassified images or objects. However, when using the loss definition from Equation 19, the loss per image can be larger than one, resulting in a performance indicator $e_t > 1$ (this usually happens especially during the first part of the training process). Another problem is the fact that using the classical expression of the importance of a stump results in negative weights for values of $e_t > 0.5$. This makes sense for classification problems: when an object is more than half of the time classified wrongly by a decision stump, the model would perform better to do opposite of what the stump says. However, in our object detection model this would restrict us to only two classes of objects without major changes in the model. Besides that, the loss function as was specified in Equation 19 does not allow this simple explanation for losses larger than 0.5, as this function consists of not only a classification error L_{cls} , but also the error on the location of the anchor box L_{reg} . Instead, we propose an importance function of the functional form $\frac{1}{e_t}$, as this solves both of the problems. Now the importance is defined for values of $e_t > 1$ and the importance is always positive. To overcome too large priority for cases when e_t approaches 0, we give a small translation to the left, ending up with the function $\alpha_t = \frac{1}{e_t+0.3}$ (see Figure 17).

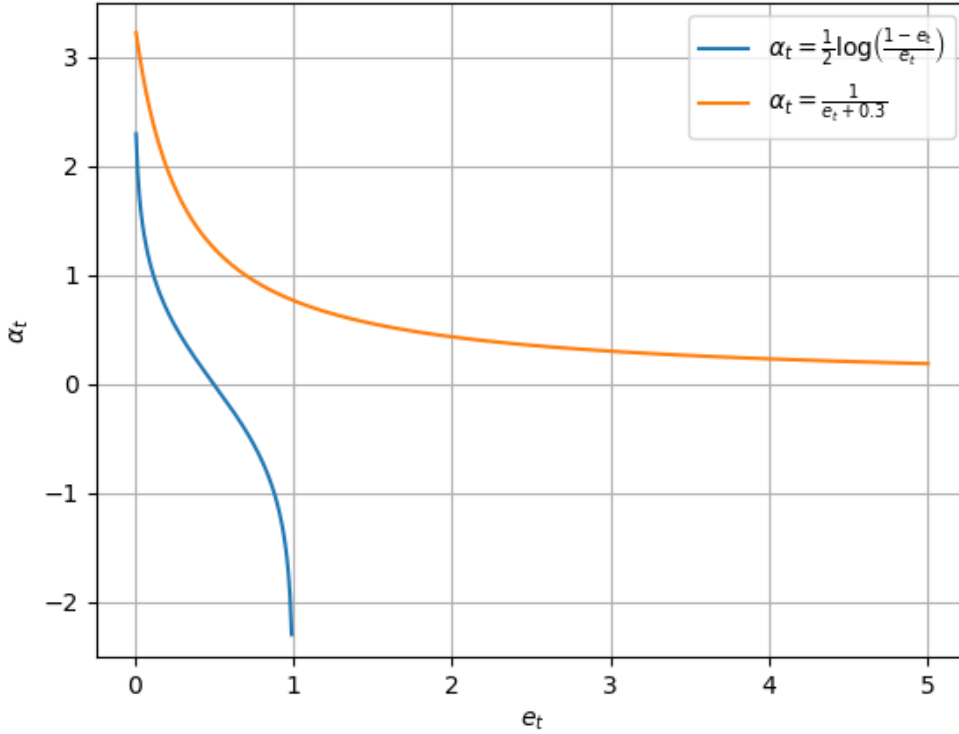


Figure 17: Comparison of the original importance weight of a stump (blue) with the new function (orange). As can be seen, the domain now also includes values of the error term above 1 and the output weight is strictly positive.

The final change compared to the original AdaBoost algorithm is the weight updating of the observations. Note in line 6 of Algorithm 1 that for every learner the error weights are normalised such that $\sum_{i=1}^N D_{t,i} = 1$. There are a few adaptations we need to make to this line of code. First of all, we only use at most 1,000 distinct images. We write down the chosen image slices for learner t as $\mathcal{S}_t \subset \mathcal{N}$, with \mathcal{N} the total set of N data points. Then we assure that the error weights of all 6,886 images always sum up to 1 by scaling the new weights D_t of the chosen samples for weak learner t such that $\sum_{i \in \mathcal{S}_t} D_{t,i} = \sum_{i \in \mathcal{S}_t} D_{t-1,i}$. That is, we effectively only redistribute the sample weight from the samples that are chosen in the 1,000 images of this iteration. Also, when calculating the new error weights we change the term $\exp(-\alpha_t y_i h_t(x_i))$ (line 6 in Algorithm 1) to $\exp(\alpha_t L_{t,i})$. As our loss term $L_{t,i}$ is not binary but rather a sum of losses composed of both classification errors and bounding box regression errors in the image, it is not useful to distinct between ‘positive’ and ‘negative’ cases (as stated previously). Instead, we can directly use the size of $L_{t,i}$ to determine the sample weight for the next iteration.

Together this leads to the following adjusted AdaBoost algorithm:

Algorithm 2 Adjusted AdaBoost; every learner is trained on a subset \mathcal{S}_t of the N data points. $D_{t,i}$ is the error weight of data point i for learner t , α_t the importance of learner t , reflecting how well the learner performs on the train data. $L_{t,i}$ is the loss on data point i for learner t

```

1: for  $i$  from 1 to  $N$  do  $D_{0,i} = \frac{1}{N}$ 
2: for  $t$  from 1 to  $T$  do
3:   Train the model (often a decision stump) on all  $N$  data points simultaneously to minimise the weighted
      fraction of misclassified training data points  $e_t$  with  $e_t = \frac{1}{1,000} \sum_{i \in \mathcal{S}_t} L_{t,i}$  with  $L_{t,i}$  defined as  $L_i$ 
      (Equation 19) in the current iteration  $t$ 
4:   Calculate the importance  $\alpha_t = \frac{1}{e_t + 0.3}$ 
5:   for  $i$  from 1 to  $N$  do
6:     Calculate the new error weights for the next iteration
7:     if  $i \in \mathcal{S}_t$  then
8:        $D_{t,i} = \frac{D_{t-1,i} \exp(\alpha_t L_{t,i})}{\sum_{i \in \mathcal{S}_t} D_{t-1,i} \exp(\alpha_t L_{t,i})} \sum_{i \in \mathcal{S}_t} D_{t-1,i}$ 
9:     else
10:       $D_{t,i} = D_{t-1,i}$ 

```

Implementation

During testing of the base model, the RPN part would normally output 300 region proposals via the NMS layer towards the CLS part. For the boosting model we will ensemble the region proposals of all T learners, leading to $T \times 300$ region proposals. These proposals are combined in a new NMS layer, which again outputs only 300 region proposals towards the CLS part of the Faster R-CNN model. This way, the best region proposals of all T learners are preserved while locations with too much overlap resulting from separate learners are filtered. When an image during an epoch has been drawn multiple times, we use the loss $L_{t,i}$ related to the final draw of that image in the epoch to calculate the new error weight. The reason for this is that this loss term is most closely related to the state of the model when calculating the error weights.

Note that this boosting algorithm focuses on the RPN part of the model. When combining the RPN part of the model with the CLS network we make three different setups. First of all we will try only to train the RPN part of the model, while continuing every epoch with both the weights β and the error weights e from the last epoch. We use a separately trained CLS layer from the base model. Secondly, we train both RPN and CLS to see how much the choice of the CLS-head is of an influence. Thirdly, we only transfer the error weights e from each epoch to the next, while resetting the weights β every epoch. This setting most closely resembles the situation that is normally present with boosting, as each individual learner is kept ‘weak’ by resetting the weights every epoch. For this third variant, we both try training the CLS each epoch and

using transferred CLS weights. We keep the number of learners constant at $T = 10$, which coincides with the number of epochs the model without boosting (base model) will be trained (see Section 5). These three variants can be summarised as follows:

- Variant 1: Training the RPN for 10 epochs while updating the importance of each slice every epoch, such that slices that causes most loss are reoccurring more in the following epoch(s).
- Variant 2: Training both the RPN and CLS for 10 epochs, using an importance function that incorporates the losses made in both parts of the model as well.
- Variant 3: Training the RPN 10 times 1 epoch in serial boosts, every time starting with the same weights but with updated importance of each slice based on the loss generated by that slice in the previous boost.

In Figure 18 these three variants are visualised.

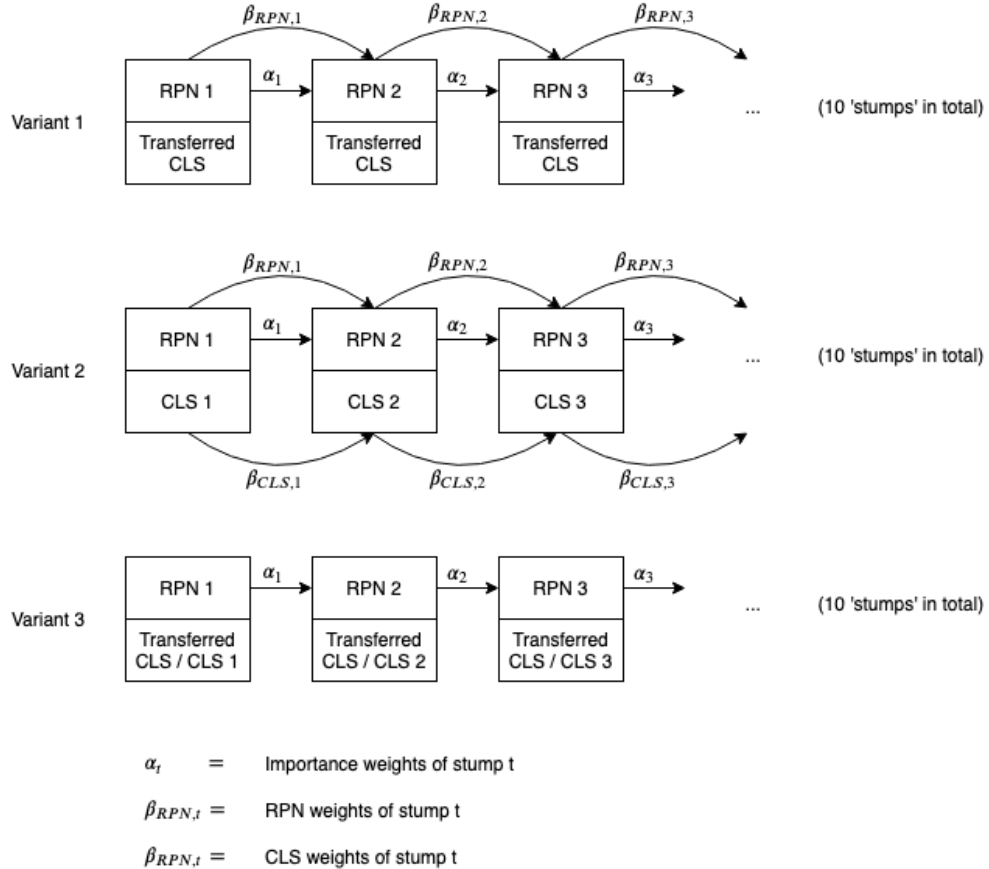


Figure 18: The three variants of transferring weights during training of the ‘stumps’.

Considering the implementation of the boosting method in a Faster R-CNN model, we have used the Faster R-CNN model by kbardool (2017).

References

- Ball, Geoffrey H and David J Hall (1967). “A clustering technique for summarizing multivariate data”. In: *Behavioral Science* 12.2, pp. 153–155.
- Bergstra, James et al. (2006). “Aggregate features and AdaBoost for music classification”. In: *Machine Learning* 65.2-3, pp. 473–484.
- Bjorgo, Einar (2000). “Using very high spatial resolution multispectral satellite sensor imagery to monitor refugee camps”. In: *International Journal of Remote Sensing* 21.3, pp. 611–616.
- Chellapilla, Kumar, Sidd Puri, and Patrice Simard (2006). “High performance convolutional neural networks for document processing”. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.
- Ciresan, Dan Claudiu et al. (2011). “Flexible, high performance convolutional neural networks for image classification”. In: *Twenty-Second International Joint Conference on Artificial Intelligence*, pp. 1237–1242.
- Comaniciu, Dorin and Peter Meer (2002). “Mean shift: A robust approach toward feature space analysis”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 5, pp. 603–619.
- Eggert, Christian et al. (2017). “A closer look: Small object detection in faster R-CNN”. In: *2017 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, pp. 421–426.
- Freund, Yoav and Robert E Schapire (1995). “A decision-theoretic generalization of on-line learning and an application to boosting”. In: *European Conference on Computational Learning Theory*. Springer, pp. 23–37.
- Giada, S et al. (2003). “Information extraction from very high resolution satellite imagery over Lukole refugee camp, Tanzania”. In: *International Journal of Remote Sensing* 24.22, pp. 4251–4266.
- Girshick, R (2015). “Training R-CNNs of various velocities”. In: *ICCV 2015 Tutorial on Tools for Efficient Object Detection*.
- Girshick, Ross (2015). “Fast R-CNN”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1440–1448.
- Girshick, Ross et al. (2014). “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587.
- Hahmann, Stefan and Dirk Burghardt (2013). “How much information is geospatially referenced? Networks and cognition”. In: *International Journal of Geographical Information Science* 27.6, pp. 1171–1189.
- Han, Xiaobing et al. (2017). “Scene classification based on a hierarchical convolutional sparse auto-encoder for high spatial resolution imagery”. In: *International Journal of Remote Sensing* 38.2, pp. 514–536.

- Haralick, Robert M, Karthikeyan Shanmugam, et al. (1973). “Textural features for image classification”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 6, pp. 610–621.
- Hastie, Trevor et al. (2009). “Multi-class AdaBoost”. In: *Statistics and Its Interface* 2.3, pp. 349–360.
- He, Kaiming et al. (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.
- Hubel, David H and Torsten N Wiesel (1959). “Receptive fields of single neurones in the cat’s striate cortex”. In: *The Journal of physiology* 148.3, pp. 574–591.
- Ioffe, Sergey and Christian Szegedy (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167*. (Visited on 03/25/2020).
- kbardool (2017). *keras-frcnn*. <https://github.com/kbardool/keras-frcnn>. (Visited on 07/18/2019).
- Keskar, Nitish Shirish and Richard Socher (2017). “Improving Generalization Performance by Switching from Adam to SGD”. In: *arXiv preprint arXiv:1712.07628*. (Visited on 03/25/2020).
- Kingma, Diederik P and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*. (Visited on 03/25/2020).
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “ImageNet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*, pp. 1097–1105.
- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). “Deep learning”. In: *nature* 521.7553, p. 436.
- LeCun, Yann, Bernhard E Boser, et al. (1990). “Handwritten digit recognition with a back-propagation network”. In: *Advances in neural information processing systems*, pp. 396–404.
- LeCun, Yann, Bernhard Boser, et al. (1989). “Backpropagation applied to handwritten zip code recognition”. In: *Neural Computation* 1.4, pp. 541–551.
- LeCun, Yann, Léon Bottou, et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.
- Lin, Tsung-Yi et al. (2017). “Focal loss for dense object detection”. In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2980–2988.
- Liu, Wei et al. (2016). “SSD: Single shot multibox detector”. In: *European Conference on Computer Vision*. Springer, pp. 21–37.
- Long, Jonathan, Evan Shelhamer, and Trevor Darrell (2015). “Fully convolutional networks for semantic segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3431–3440.
- Luo, Liangchen et al. (2019). “Adaptive gradient methods with dynamic bound of learning rate”. In: *arXiv preprint arXiv:1902.09843*. (Visited on 03/25/2020).
- Rawat, Waseem and Zenghui Wang (2017). “Deep convolutional neural networks for image classification: A comprehensive review”. In: *Neural Computation* 29.9, pp. 2352–2449.

- Redmon, Joseph et al. (2016). “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 779–788.
- Ren, Shaoqing et al. (2015). “Faster R-CNN: Towards real-time object detection with region proposal networks”. In: *Advances in Neural Information Processing Systems*, pp. 91–99.
- Rezatofighi, Hamid et al. (2019). “Generalized intersection over union: A metric and a loss for bounding box regression”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 658–666.
- Schapire, Robert E (1990). “The strength of weak learnability”. In: *Machine Learning* 5.2, pp. 197–227.
- Simard, Patrice Y, David Steinkraus, John C Platt, et al. (2003). “Best practices for convolutional neural networks applied to visual document analysis”. In: *Seventh International Conference on Document Analysis and Recognition (ICDAR)*. Vol. 3. 2003, pp. 958–963.
- Stahl, Tobias, Silvia L Pintea, and Jan C van Gemert (2019). “Divide and Count: Generic Object Counting by Image Divisions”. In: *IEEE Transactions on Image Processing* 28.2, pp. 1035–1044.
- UN High Commissioner for Refugees (UNHCR) (June 22, 2018). *Global Trends: Forced Displacement in 2017*. UN High Commissioner for Refugees (UNHCR). URL: <https://www.refworld.org/docid/5b2d1a867.html> (visited on 04/08/2019).
- Viola, Paul and Michael J Jones (2004). “Robust real-time face detection”. In: *International Journal of Computer Vision* 57.2, pp. 137–154.
- Wang, Shifeng, Emily So, and Pete Smith (2015). “Detecting tents to estimate the displaced populations for post-disaster relief using high resolution satellite imagery”. In: *International Journal of Applied Earth Observation and Geoinformation* 36, pp. 87–93.
- Watkins, John F and Hazel A Morrow-Jones (1985). “Small area population estimates using aerial photography”. In: *Photogrammetric Engineering and Remote Sensing* 51.12, pp. 1933–1935.
- Wen, Xuezhi et al. (2015). “A rapid learning algorithm for vehicle classification”. In: *Information Sciences* 295, pp. 395–406.
- Wieland, Marc and Massimiliano Pittore (2014). “Performance evaluation of machine learning algorithms for urban pattern recognition from multi-spectral satellite images”. In: *Remote Sensing* 6.4, pp. 2912–2939.